John Krogstie
Andersen Consulting and
The Norwegian University of Science and
Technology

Arne Sølvberg
The Norwegian University of Science and
Technology

Information Systems Engineering:

Conceptual Modeling in a quality perspective

January 2, 2000

Information Systems Engineering:
Conceptual Modeling in a quality perspective

by
John Krogstie, Andersen Consulting and NTNU
and Arne Sølvberg, NTNU

Preface

This book presents an overview of the area of conceptual modeling, which is an important area within information systems engineering. It is intended for advanced teaching within information systems engineering, that should be based upon an introductory course on the subject. Such an introductory course can for instance be based on the book *Information Systems Engineering: An Introduction* by Arne Sølvberg and David C. Kung (Springer 1993). The students should have had some experience in modeling for instance through using data modeling such as ER-modeling and process modeling such as DFD before embarking upon the course.

This book is based upon work originally presented in nine doctoral theses delivered at the information systems group at IDI, NTNU under the supervision of Professor Arne Sølvberg. The theses are:

- HICONS: A general diagrammatic framework for hierarchical modeling [344] by Guttorm Sindre.
- A prototyping approach to validation of conceptual models in information systems engineering [236] by Odd Ivar Lindland.
- COMIS: A conceptual model for information systems [404] by Mingwei Yang.
- Explanation generation in information systems engineering [151] by Jon Atle Gulla.
- Executable conceptual models in information systems engineering [398] by Geir Willumsen.
- A configuration management approach for supporting cooperative information systems development [7] by Rudolf Andersen.
- Complexity reduction in information systems modeling [331] by Anne Helga Seltveit.
- Conceptual modeling for computerized information systems support in organizations [207] by John Krogstie.
- Conceptual modeling and composition of flexible workflow models [53] by Steinar Carlsen.

Material from the separate theses and accompanying publications is in this book reorganized according to a framework for quality of conceptual models, which have been developed primarily by Carlsen, Krogstie, Lindland, Sindre, and Sølvberg in cooperation.

An outline of the book is presented in the end of the introduction chapter. The book covers a lot of subjects, many of which are only touched upon briefly. These are covered in more detail in the material referenced throughout the book. A course being based on the book would typically also include a set of up to date articles and other publications on specific areas of interest.

December 1999

John Krogstie
Arne Sølvberg

Table of Contents

List of Figures

List of Tables

1. Introduction

Conceptual models are normally constructed during the problem analysis and requirements specification process of information systems engineering. They are often used as a starting point for constructing and implementing the information system.

A conceptual model can be defined as

> a model of the phenomena in a domain at some level of approximation, which is expressed in a semi-formal or formal language.

In this text, we apply the following limitations:

- The languages for conceptual modeling are mostly diagrammatic with a limited vocabulary. The main symbols of the languages represent *concepts* such as states, processes, entities, and objects. We mostly use the terms 'phenomena' and 'phenomena classes' instead of 'concepts' in this text since the word 'concept' is used in many different meanings in natural language.
- Conceptual models are primarily used as an intermediate representation for development and maintenance of information systems. We recognize that conceptual modeling languages may be useful also for other purposes such as organizational modeling or process modeling when there is no immediate system implementation in mind. We will not treat such usage of modeling languages in detail in the book.
- The conceptual languages presented in this text are meant to have general applicability, that is, they are not made specifically for the modeling of a limited area.

We combine the use of conceptual models with the philosophical outlook that reality is socially constructed. Most of the current modeling approaches are based on an objectivistic ontology,e.g., "the real world consists of entities and relationships" [62, 203]. However, this assumption is not shared by everybody, in [348] for instance, it is focused on that what is modeled is some persons *perception* of the "real world".

The present practice of modeling includes a large element of subjectivity [232]. This subjectivity exists whether or not the data-focused approaches uses 'entities', 'objects', or 'phenomena' as main concepts. If entities *are* taken to have real-world existence, then the participants in the modeling effort must

choose from the infinitely large number of entities that exist, only those entities that are relevant and suitable for inclusion in the model. Consequently, the process of creating such a model is not value-free and the resulting conceptual model is not unbiased. If real-world existence of the relevant phenomena is *not* assumed, then the entities are, by definition, created subjectively by the participants in the modeling effort in order to understand the situation at hand. In either case, the conceptual model serves only as an interpretation of "reality".

Thus, in both cases, it will be useful to admit to this subjectivity and allow several models to co-exist, even if only one of them will be used for building the information systems. There are indeed approaches that acknowledge several realities. Some approaches are grounded in object orientation [165, 312]. Another approach is Multiview [16], which has a constructivistic worldview and uses traditional conceptual languages as an important part of the methodology. A similar attempt to integrate Soft system methodology (SSM) [61] and software engineering approaches is reported in [97].

Even if traditional conceptual modeling languages may support a constructivistic worldview, they usually do not have explicit constructs for capturing differing views directly in the model and making these available to those who use the model. They neither have the possibility to differentiate between the rules of necessity and deontic rules.

Conceptual modeling languages are biased towards a particular way of perceiving the world:

- The languages have constructs that force both analysts and users to emphasize some aspects of the world and neglect others.
- The more the analysts and users work with one particular language, the more their thinking will be influenced by this, and their awareness of those aspects of the world that do not fit in may consequently be diminished recall the Sapir-Whorf hypothesis which states that a person's understanding of the world is influenced by the language he uses [354].
- For the types of problems that fit well with the approach, neglecting features that are not covered may even have a positive effect, because it becomes easier to concentrate on the relevant issues. However, it is hard to know what issues are relevant. Different issues within a problem situation may be relevant for different people at the same time.

1.1 Organizations are Outcomes of Social Construction

Organizations are constantly under the pressure of *change* from internal as well as external forces. Most organizations are supported by a *portfolio of application systems* that likewise have to be changed, often rapidly, for the organization to be able to keep up and extend its activities. The portfolio usually consists of a set of individual, but highly integrated application systems whose long-term evolution should be coordinated as a whole.

Organizational change may be viewed from different philosophical points of view. Two common sets of assumptions are the objectivistic belief system and the constructivistic belief system [149]. They may be distinguished through differences in ontology (what exists that can be known), epistemology (what relationship is there between the knower and the known), and methodology (what are the ways of achieving knowledge).

Organizations are made up of individuals who perceive the world differently from each other. The constructivistic view is that an organization develops through a process of *social construction*, based on its individuals' perception of the world. In the objectivistic view [149] there exists only one reality, which is measurable and essentially the same for all. The objectivistic belief system can simplistically be said to have the following characteristics:

- The ontology is one of realism, asserting that there exist a single reality which is independent of any observer's interest in it and which operates according to immutable natural laws. Truth is defined as that set of statements whose natural or intended model are isomorphic to reality.
- The epistemology is one of dualistic objectivism, asserting that it is possible, indeed mandatory, for an observer to exteriorize the phenomenon studied, remaining detached and distant from it and excluding any value considerations from influencing it.
- The methodology is one of interventionism, stripping context of its contaminating influences so that the inquiry can converge on truth and explain the things studied as they really are and really work, leading to the capability to predict and to control.

The constructivistic belief system has the following characteristics according to [149]:

- The ontology is one of relativism, asserting that there exist multiple socially constructed realities ungoverned by any natural laws, causal or otherwise. "Truth" is defined as the best-informed and most sophisticated construction on which there is agreement.
- The epistemology is subjectivistic, asserting that the inquirer and the inquired-into are interlocked in such a way that the findings of an investigation are the literal creation of the inquiry process.
- The methodology is hermeneutical and involves a continuing dialectic of iteration, analysis, critique, reiteration, reanalysis, and so on, leading to the emergence of a joint construction and understanding among all the stakeholders.

Many features of the constructivistic paradigm have emerged from "hard natural sciences such as physics and chemistry. The argument for the new paradigm can be made even more persuasively when the phenomena being studied involve human beings, as in the "soft social sciences. Much of the theoretical discussion in the social sciences is at present dedicated to analyzing constructivism and its consequences [75]. The idea of reality construction has

been a central topic for philosophical debate during the last two decades, and has been approached differently by French, American, and German philosophers.

Many different approaches to constructivistic thinking have appeared, although probably the most influential one is that Berger and Luckmann [26]. Their insights will be used as our starting point. Their view of the social construction of reality is based on Husserl's phenomenology. Whereas Husserl was primarily a philosopher, Schutz [326] took phenomenology into the social sciences. From there on it branched into two directions: Ethnomethodology, primarily developed by Garfinkel [130], and the social constructivism of Berger and Luckmann. Whereas ethnomethodology is focused on questioning what individuals take as given in different cultures, Berger and Luckmann developed their approach to investigate *how* these presumptions are constructed.

Organizations are realities constructed socially through the joint actions of the social actors in the organization [136], as illustrated in Fig. 1.1.

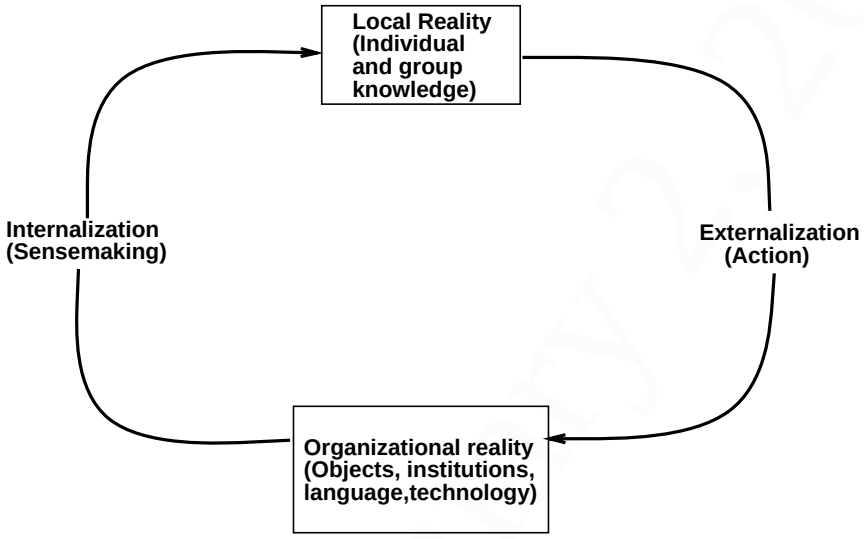


Fig. 1.1. Social construction in an organization

An organization consists of individuals who view the world in their own specific way, because each of them has different experiences arising from work and other activities. The *local reality* refers to the way an individual perceives the world in which he or she acts. The local reality is the way the world is for the individual; it is the everyday perceived reality of the individual social actor. Some of this local reality may be made explicit and talked about. However, a lot of what we know is *tacit*. When the social actors of an organization

act, they *externalize* their local reality. The most important ways in which social actors externalize their local reality are by speaking and constructing languages, artifacts, and institutions. What they do, is to construct *organizational reality* by making something that other actors have to relate to as being part of the organization. This organizational reality may consist of different things, such as institutions, language, artifacts, and technology. Finally, *internalization* is the process of making sense out of the actions, institutions, artifacts etc. in the organization, and making this organizational reality part of the individual local reality. This linear presentation does not mean that the processes of externalization and internalization occur in a strict sequence. Externalization and internalization may be performed simultaneously. Also, it does not mean that only organizational reality is internalized by individuals. Other externalizations also influence the construction of the local reality of an individual.

Changing the computerized information system (CIS) support in an organization, for instance by introducing new application systems may also be looked upon as a process of social construction. This view has received increasing interest in the information systems community in recent years [121, 235, 340, 400]. This outlook is adopted in this book, especially when focusing on the creation and maintenance and further development of conceptual models in connection with improving the computerized information system support of an organization. This does not mean that we are ignorant of the more technical aspects of computerized information systems support. Constructed realities are often related to, and also often inseparable from, tangible phenomena.

1.2 Conceptual Modeling as Social Construction

The construction of a conceptual model of "reality" as it is perceived by someone is partly a process of externalization of parts of this person's internal reality, and will in the first place act as organizational reality for the audience of the model. This model can then be used in the sense-making process by the other stakeholders, internalizing the views of the others if they are found appropriate. This internalization is based on pre-understanding, which includes assumptions implicit in the languages used for modeling. The language in turn is learned through internalization.

After reaching a sufficiently stable shared model one might wish to externalize this in a more material way, transferring it to the organization in the form of computer technology. Here a new need for internalization of the technology is needed for the CIS to be useful for the part of the organization that is influenced by it. Also here, it should be possible to utilize the conceptual models to understand what the CIS does, and most importantly, *why* it does it. Making sense of the technology is important to be able to change it,

and the conceptual models already developed can act as a starting point for additional maintenance efforts on the CIS when deemed necessary.

It should be noted that the abilities and opportunities for the different social actors in the organization to externalize their local reality will differ in several ways. Since the languages and types of languages used are often predefined when a decision to create an application system is made, persons with long experience in using these kinds of languages will have an advantage in the modeling process. This applies especially to the specialists on computer technology. This is not necessarily bad, for if they did not have this knowledge it would not be interesting to include them in the development process in the first place. Rather, it is important to be aware of this difference, to avoid the most apparent dangers of model monopoly as discussed by Bråten [38]. What is also apparent is that some persons in the organization have a greater possibility to externalize their reality than others, both generally (the financiers of an endeavor will for instance usually be in a position to bias a solution in their perceived favor) and specifically, by the use of certain modeling techniques. Gjersvik has for instance investigated how the way management perceive the world can be more easily externalized in a CIS than the way shop-floor workers perceive the world [136].

The use of conceptual models constructed as part of the development and maintenance of application systems has been discussed by several researchers [44, 93, 186, 217, 393]. This discussions can be summarized as follows:

- Representation of systems and requirements: The conceptual model represents properties of the problem area and perceived requirements for the information system. A conceptual model can give insight into the problems motivating the development project, and can help the systems developers and users understand the application system to be built. Moreover, by analyzing the model instead of the business area itself, one might deduce properties that are difficult if not impossible to perceive directly since the model enables one to concentrate on just a few aspects at a time.
- Vehicle for communication: The conceptual model can serve as a means for sense-making and communication among stakeholders. By hopefully bridging the realm of the end-users and the CIS, it facilitates a more reliable and constructive exchange of opinions between users and the developers of the CIS, and between different users. The models both help and restrict the communication by establishing a nomenclature and a definition of phenomena in the modeling domain.
- Basis for design and implementation: The conceptual model can act as a prescriptive model, to be approved by the stakeholders who specify the desired properties of a CIS. The model can establish the content and boundary of the area under concern more precisely. During design and implementation of the CIS, the relevant parts of the model guide the development process. Similarly, the design and implementation might afterwards

be tested against the model to make sure that the different representations are consistent. When the model is formal and contains sufficient detail, it is often possible to produce the application system more or less directly from the model.
- Documentation and sensemaking: The conceptual model is an easily accessible documentation of the CISs that are in use in the organization. Due to its independence of the implementation, it is less detailed than other representations, while still representing the basic functionality of the system. Compared to manually produced textual documentation, the conceptual model is easier to maintain since it is constructed as part of the process of developing and maintaining the application system in the first place. With the introduction of more flexible methodologies and tool support, conceptual models are also likely to be used in reverse engineering and re-engineering, and when reusing artifacts constructed in connection with other application systems.

Summing up, a conceptual model is used both for communication and representation, and faces demands from both social and technical actors. As a consequence of this duality, requirements for conceptual modeling languages and modeling techniques will pull in opposite directions.

1.3 Quality Dimensions of Conceptual Modeling

We have organized this book according to a framework that has been developed for understanding the quality of conceptual modeling and modeling languages. The main structure of this framework, as originally presented in [239], is illustrated in Fig. 1.2. The basic idea is to evaluate the quality of models along three dimensions by comparing sets of statements. These sets are:

- $\mathcal{M}$, the externalized model, that is, the set of all the statements explicitly or implicitly made in the model.
- $\mathcal{L}$, the language extension, that is, the set of all statements which can be made according to the vocabulary and grammar of the modeling languages used.
- $\mathcal{D}$, the modeling domain, that is, the set of all statements that can be stated about the problem at hand.
- $\mathcal{I}$, the audience interpretation, that is, the set of all statements which the audience (i.e. various actors in the modeling process) think that the model comprises.

Model quality is defined using the relationships between the model and the three other sets:

- *Syntactic quality* is the degree of correspondence between model and language extension, that is, the set of syntactic errors is $\mathcal{M} \setminus \mathcal{L}$.

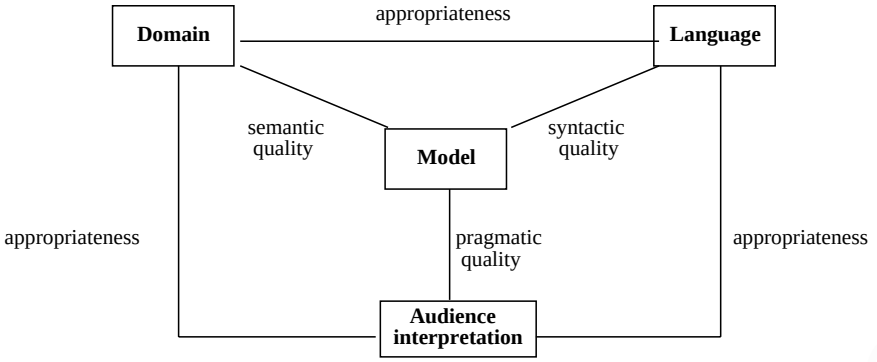


Fig. 1.2. The model quality framework of Lindland et al. (From [239])

- *Semantic quality* is the degree of correspondence between model and domain. If $\mathcal{M} \setminus \mathcal{D} \neq \emptyset$ the model contains invalid statements; if $\mathcal{D} \setminus \mathcal{M} \neq \emptyset$ the model is incomplete. Since total validity and completeness are generally impossible, the notions of *feasible validity* and *feasible completeness* are employed.
- *Pragmatic quality* is the degree of correspondence between model and audience interpretation (i.e., the degree to which the model has been understood). If $\mathcal{I} \neq \mathcal{M}$, the comprehension of the model is not completely correct. Usually, it is neither necessary nor possible that the whole audience understand the entire conceptual model – instead, each group in the audience should understand the part of the model which is relevant to them. *Feasible comprehension* is defined along the same lines as feasibility for validity and completeness.

In addition to these primary quality concerns, the correspondence between domain and language, between domain and audience interpretation, and between language and audience interpretation may affect the model quality indirectly. These relationships are all denoted *appropriateness* Fig. 1.2. We also differentiate between the quality goals and means for achieving these goals.

1.4 Outline of the Book

The quality framework is presented in more detail in Chap. 3. In Chap. 2 we first give an overview of the different mechanisms and perspectives used in conceptual modeling. Means to support quality on different levels are then discussed in Chapters 4– 7. Aspects of a modeling methodology based on the use of the framework is outlined in Chap. 8. An example on how to apply the framework to evaluate a modeling approach is given in Appendix A.

Mathematical symbols are explained in Appendix C, and the terminology is explained in Appendix D.

The main example used throughout the book concern an information system for supporting the arrangement of a scientific conference. It is a variant of the widely used "IFIP working conference" example which was originally presented in 1982 [285] and has been used since then to illustrate a variety of information system modeling approaches.

In this book, the example describes a real situation, since we have been developing a system of this sort using some of the described languages. Thus, the case description is not the traditional one, but is taken from an actual project where we developed, used and maintained a support system for the IFIP WG8.1 Working Conference on Information Systems development for Decentralized Organizations (ISDO95), which took place in Trondheim, Norway, in August 1995 [349].

IFIP is the acronym for International Federation for Information Processing. An IFIP working conference is an international conference that provides an opportunity for the computer scientists from IFIP member countries to discuss and interchange research results and new ideas on selected research fields.

The management of such a conference is usually done by two cooperating committees. The *program committee* (PC) handles the contents of the conference, say, the reviewing of papers, comprising sessions and tutorials, etc. The *organizing committee* (OC) handles the administration work, e.g. sending out invitations, registration of attendants, arranging time and places for sessions, dealing with financial matters, etc.

We will return to this and other examples throughout the book, adding more detail to the case description as needed.

2. Conceptual Modeling Languages

In this chapter, we will give an overview of mechanisms and perspectives used in conceptual modeling. We will first look upon modeling in general as hierarchical abstraction. Then we will present different modeling languages according to the main phenomena they describe, and discuss the usefulness of the possibility of applying several such perspectives at the same time in an integrated manner. An approach supporting all the described perspectives, PPP, is presented in the end of the chapter.

2.1 Modeling as Hierarchical Abstraction

A conceptual model is an abstraction. One mechanism for abstraction used in many of the existing languages for conceptual modeling is the use of *hierarchies*. The importance of hierarchical abstractions is based on the following assumptions.

- Hierarchies are essential for human understanding of complex systems.
- Thinking in terms of hierarchical constructs such as aggregation and generalization appears to be very natural.
- Information systems are complex systems because they must reflect the part of the world they process information about,
- A proper support for hierarchical constructs is an essential requirement throughout the entire information system development and maintenance process.

2.1.1 What Is a Hierarchy?

Here we will discuss what a hierarchy is in more detail. The first subsection discusses the possibilities for arriving at a precise definition of the term 'hierarchy' in terms of graph theory. As will be seen, however, it is difficult to come up with a definition which is precise and at the same time satisfactory. The second subsection thus argues that being hierarchical is very much a question of degree.

Hierarchical: A Question of Definition. In [147, 341] a hierarchy is defined rather vaguely as any collection of Chinese boxes (where each box can contain several smaller boxes). [261] refrains from giving any exact definition of what a hierarchy is, but lists some properties which all hierarchies should have, namely "vertical arrangement of subsystems which comprise the overall system, priority of action or right of intervention of the higher level subsystems, and dependence of the higher level subsystems upon the actual performance of the lower levels. More precise definitions are given in [17] and [49].

Some works also identify different kinds of hierarchical systems. [17] distinguishes formally between *division hierarchies* and *control hierarchies*. [261] operates with three notions of hierarchical levels, namely *strata* (levels of description or abstraction), *layers* (levels of decision complexity), and *echelons* (organizational levels). All in all it seems that the word "hierarchy" may be used in rather different ways by different authors — as stated in [352] some use it indiscriminately for any partial ordering, whereas the above definitions require something more.

It is difficult to come up with a strict and precise definition distinguishing hierarchical systems from other systems. However, since it is important to make clear what we are talking about, we need some kind of definition of what a hierarchy is.

To this end it is illuminating to look at the definition presented by Bunge in [49]:

> **H** is a hierarchy if and only if it is an ordered triple **H** = <**S**,**b**,**D**> where **S** is a nonempty set, **b** a distinguished element of **S** and **D** a binary relation in **S** such that
>
> 1. **S** has a single beginner, **b**. (That is, **H** has one and only one supreme commander.)
> 2. **b** stands in some power of **D** to every other member of **S**. (That is, no matter how low in the hierarchy an element of S may stand, it is still under the command of the beginner.)
> 3. For any given element **y** of **S** except **b**, there is exactly one other element **x** of **S** such that **Dxy**. (That is, every member has a single direct boss.)
> 4. **D** is transitive and antisymmetric.
> 5. **D** represents (mirrors) domination or power. (That is, **S** is not merely a partially ordered set with a first element: the behavior of each element of **S** save its beginner is ultimately determined by its superiors.)

As pointed out by Bunge, this definition does two things:

– The first four points state what a hierarchy is in a graph-theoretic sense, namely a strict tree-structure.[1]
– The fifth point introduces an extra requirement on the nature of the relations (i.e. edges) between the nodes, namely that they represent domination or power.

Thus, Bunge makes the important point that whether something is a hierarchy or not cannot be determined by graph-theoretic considerations alone. However, Bunge's definition might be a little too strict:

– the graph-theoretic demands are very limiting. In real life it often happens that a node can have more than one boss, or even that there are cycles in the graph, and still many people might consider the system to be of a hierarchical nature.
– the requirement that nodes are related by domination severely limits the scope of hierarchical systems — as stated by Bunge himself reciprocal action, rather than unidirectional action, seems to be the rule in nature (which leads Bunge to the conclusion that it is misleading to speak of hierarchies in nature: "Hierarchical structures are found in society, e.g. in armies and in old-fashioned universities; but there are no cases of hierarchy in physics or in biology"). Since one might want to be able to model practically anything, we have to recognize other kinds of hierarchical relations in addition to domination or power.

To achieve more generality, we will allow more general graphs to be considered as hierarchical systems. But it will also be useful to have a specific term for those systems which satisfy the rather restrictive requirements stated above. Below we will use the following terminology:

– Strictly hierarchical graph: a digraph whose underlying graph is a tree, and for which there is one specific vertex **b** from which all other vertices can be reached (this is the distinguished element of Bunge's definition).
– Weakly hierarchical graph: a connected acyclic digraph which deviates from the former in that there is no distinguished element and/or in that its underlying graph is cyclic. Mathematically, this class of graphs are called DAGs (directed acyclic graphs).
– Cyclic hierarchical graph: a cyclic digraph.

Obviously, the latter two notions should be used carefully – there is no point in calling any graph a hierarchy. Thus, even if we allow some DAGs, and maybe even some cycles, we should still require that a graph is pretty close to being a *strict* hierarchy if we call it hierarchical.

[1] To be precise, it is an open-ended directed graph whose *underlying* graph (i.e. the undirected parallel of a directed graph) is a tree, since trees, graph-theoretically, are undirected graphs. For an introduction to graph theory, including definitions of graphs (directed and undirected), trees, and underlying graphs, see for instance [399].

The meaning of our suggested terminology can be visualized by Fig. 2.1.

Of these graphs (a) would not be a hierarchy because it is not connected (but it might be two hierarchies), and (b) would not be a hierarchy because the edges are not directed. (c) on the other hand, is the kind of graph which satisfies Bunge's requirements, i.e. it is a strict hierarchy. (d) would not be accepted as a hierarchy according to Bunge's definition because the underlying graph has a cycle (i.e. the middle element at the lowest level has two bosses), but we might call it a weak hierarchy. Similarly, (e) does not have one distinguished element – there are two elements on top which do not control each other. This could also be a weak hierarchy in our terminology. Finally, (f) contains a cycle and is thus clearly excluded by Bunge's definition, whereas we could call it a cyclic hierarchy (because although containing a cycle, the graph is not very far from being a strict hierarchy).

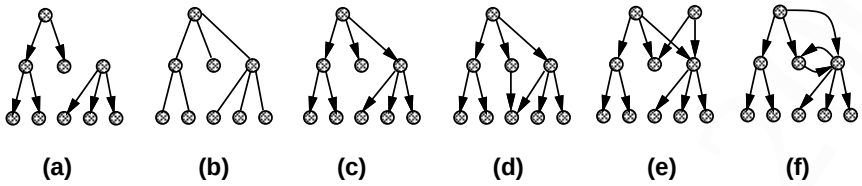


Fig. 2.1. Six graphs

The motivation for removing some of the restrictions of Bunge's definition is that we want to be as general as possible, and clearly many people might feel that systems are hierarchical even when they are not strictly hierarchical. This is exemplified by the two graphs of Fig. 2.2, where (a) breaks the single boss requirement, and (b) breaks the antisymmetry requirement. If the edges denote the relation like "is the boss of"[2], it is still likely that both systems will be considered as hierarchical. Moreover, our definition has not required that the relations denoted by the edges be transitive. Clearly, most hierarchical relations are transitive (e.g. if A is the boss of B, and B the boss of C, it is also true that A is the boss of C), but there is no point in rejecting cases where this does not apply (e.g. if A is the parent of B, and B the parent of C, it will not be true to say based on this that A is the parent of C, and still people might feel that "parent of" is a typically hierarchical relation).

Having loosened up Bunge's graph-theoretic restrictions it might seem that we may end up calling any kind of directed graph a hierarchy. However, this is not our intention. We still need some requirement corresponding to the fifth point of Bunge's definition. *Dominance* or *power* is too narrow. Still we need to make some restriction on the semantics of the relation denoted by the

[2] In (b) Bo and Dan might for instance supervise two different business areas, both working on both.

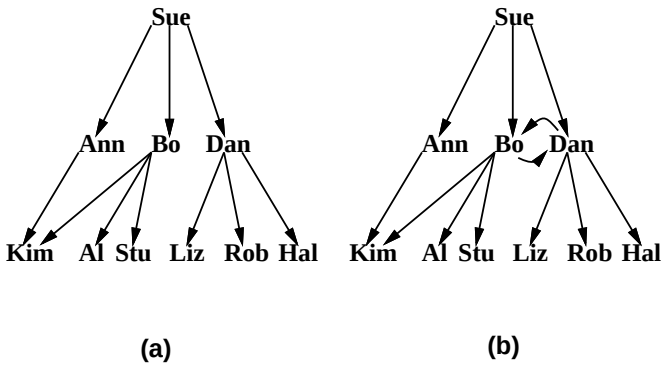


Fig. 2.2. Two graphs with hierarchical tendencies

edges. This is not easy, and can only be dealt with when we have discussed in the next subsection what it means to be more or less hierarchical.

Hierarchical: a Question of Degree. If one takes everything into consideration, a model of a situation will be a general graph rather than a hierarchy. Depicting something as a strict hierarchy will therefore be a simplification, and this simplification may be more or less appropriate, depending on the *distance* between the hierarchy depicted and the actual situation as it is perceived.

How can such *distance* be measured? Given a general connected digraph, how would you answer the question "How close is this graph to being a hierarchy?" From a general connected digraph, a strict hierarchy can be obtained by cutting some edges, so a first attempt could be to count how many edges one would have to cut, or rather the ratio of cut edges to the total number of edges. With this approach we would say that the digraph of Fig. 2.3(a) is obviously closer to being hierarchical than the one of (b), since in (a) we have to cut only 2 out of 10 edges, whereas in (b) we have to cut 4 out of 10. However, the soundness of this kind of computation relies on the assumption that all links are equally important, which need of course not be true. To deal with other cases, each edge must be assigned a weight, signaling its *importance*. In Fig. 2.4 weights have been assigned, and now it is (b) which is closest to being a hierarchy, because the minimum cut has a total of only 5 weights, whereas the same number for (a) is 12.

If the given relation is "is the boss of" weights should reflect the degree of influence that the supervisor has over the subordinate; if the graphs are call graphs for a software system, importance will depend on the frequency of calls. Generally, importance is a very problematic notion, and we will not enter any further discussions of it here. Instead we will only conclude that:

– A general directed graph can be more or less hierarchical
– Its closeness to a strict hierarchy is dependent on:

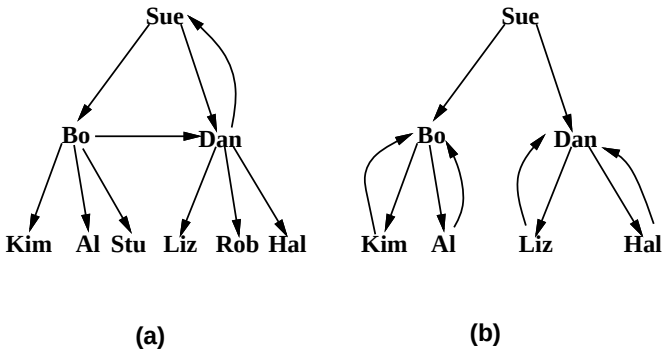


Fig. 2.3. Two general digraphs

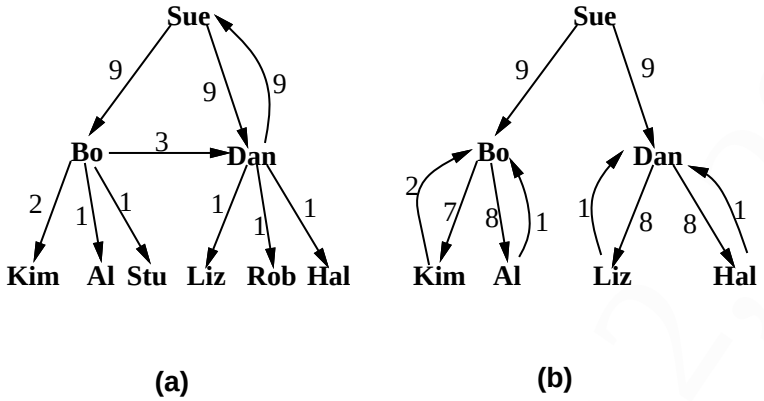


Fig. 2.4. Two weighted digraphs

- The structure of the graph.
- The importance of individual connections.

Being hierarchical is thus a question of degree. Not only specific graph models of the real world can be evaluated according to this; we can also compare different kinds of relations. Obviously, some relations, like "is the boss of", tend to result in rather hierarchical graphs, whereas for instance "loves" is not likely to do so.

In Fig. 2.5, (a) is a plausible picture of "who is the boss of who in Dept. X" and (b) is a plausible picture of "who loves who in Dept. X". As can be seen, (a) is almost a strict hierarchy, whereas (b) is not even connected (and thus very far from being a hierarchy). That the "who loves who in Dept. X" should form a hierarchy, like in (c), seems pretty unlikely because a relation like "loves" is inherently non-hierarchical (as opposed to for instance "is the boss of"). Consequently, even if the situation in (c) occurred, one might not feel that this is a hierarchy. Thus,

– Some kinds of relations are hierarchical of nature, and others are not.
– For the former it might be interesting to simplify the presentation of some knowledge by cutting edges to obtain hierarchies.
– For the latter, it seems that such an approach would make no sense, as it would be confusing rather than enlightening to present them as hierarchical.

Still, we have basically only made it clear that we want to deal with more situations than what falls under the rather strict definition of Bunge – in fact we want to be able to deal with almost any situation where something like a hierarchical abstraction construct occurs. In the next section we will identify some useful constructs in this respect.

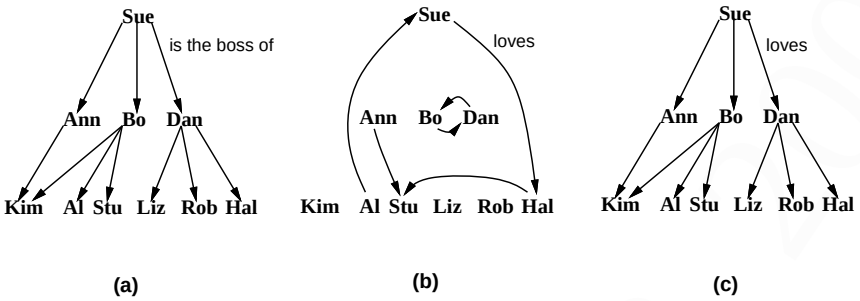


Fig. 2.5. Hierarchical and non-hierarchical relations

2.1.2 Four Standard Hierarchical Relations

There is a vast number of hierarchies that one might want to model, and these have rather diverse properties. Imagine for example organization hierarchies, definition hierarchies, goal hierarchies, file system hierarchies, and operating system process hierarchies.

Work in the field of semantic data modeling [174, 301, 307]) and semantic networks ([117]) has lead to the identification of four standard hierarchical relations:

– **classification**,
– **aggregation**,
– **association**, and
– **generalization**.

We define the following abbreviations:

CAGA $\stackrel{\text{abbr}}{=}$ classification, aggregation, generalization, and association;
AGA $\stackrel{\text{abbr}}{=}$ aggregation, generalization, and association.

The four constructs have the following definition [307]:

Classification: specific instances are considered as a higher level object type via the *is-instance-of* relationship (for example, "Rod Stewart" and "Mick Jagger" are specific instances of "singers").

Aggregation: an object is related to the components that make it up via the *is-part-of* relationship (for example, a bicycle has wheels, a seat, a frame, handlebars etc.).

Generalization: similar object types are abstracted into a higher level object type via the *is-a* relationship [3] (for example, an employee is a person).

Association: several object types are considered as a higher level set object-type via the *is-a-member-of* relationship (for example, the sets "men" and "women" are members of the set "sex-groups"). Association is also likely to be encountered under the names of *membership* (e.g. [307]), *grouping* (e.g. [174]), or *collection* (e.g. [156]).

Classification may be regarded as orthogonal to the other three – whereas the others construct bigger things from smaller things (on the same meta-level), classification results in a shift of meta-level, in accordance with the philosophical notions of *intension* and *extension* [56, 96]. The *intension* of "man" is the property of being a man, whereas the extension of "man" (in any specific world, at any specific time) will be the set of all existing men (in that specific world, at that specific time). Going one meta-level higher from "man", one can get to "species", of whose extension "man" is a member (in this particular world, at this particular time). One does not have to go much higher until there are only very abstract notions like "words" and "concepts", so it is of limited interest to use many meta-levels in a model.

For the other three constructs, the complicated notions of intension and extension are unnecessary, and rather straightforward set-theoretic definitions can be provided:

- Aggregation corresponds to the *Cartesian product*: If the set A is said to be an aggregation of the sets A_1, ... ,A_N this means that $A \subseteq A_1 \times ... \times A_N$, i.e. each element of A consists of one element from each of A_1, ... ,A_N.
- Generalization corresponds to *union*: If the set A is a generalization of the sets A_1, ... ,A_N, this means that $A \subseteq A_1 \cup ... \cup A_N$.
- Association corresponds to *membership* (i.e. embracing by set brackets): If the set A is an association of the sets A_1, ... ,A_N this means that $A = \{ A_1, ... ,A_N \}$.

Classification should not be confused with set-theoretic membership, nor the notion of class with that of set, although there are clearly similarities in both cases. A class can be viewed as a collection of its instances. Moreover, each instance can be thought of as 'a member of ' a class. However, a set is an extensional notion whose identity is determined by its membership. Thus, two

[3] Some authors use "is-a" for classification.

sets A and B are equal if they have the same members, unlike classes where equality cannot be decided by simply comparing their instances. Turning to cognitive psychology, one has identified three types of theories to explain how people develop and use categories [104]:

1. Attribute theory: Contends that one think of a list of defining attributes or features. For example, fish swim and have gills. We have in this book defined the term class according to this theory. There are some deficiencies of this approach. It is not always possible to specify defining features, and it does not take into account goodness-of-examples effects; that some instances are more typical and representative than others.
2. Prototype theory. States that when a person is presented a set of stimuli, they abstract the commonalties among the stimulus set and the abstracted representation is stored in memory. A prototype is the best representation of a category. For example, a prototypical fish might be the size of a trout, have scales and fins, swim in an ocean, lake, or river and so forth. We have a general or abstract conception of fish which somehow is typical or representative of the variety of examples with which we are familiar. When given a particular example, we compare it to the abstract prototype of the category. If it is sufficiently similar to the prototype, we then judge it to be an instance of the category.
3. Exemplar theory: Assumes that all instances are stored in memory. New instances encountered are then compared with the set of exemplar already known. This theory does not assume the abstraction of a prototype, a best example.

Parsons and Wand [298] presents some guidelines for how to decide upon classes and class structure based on the cognitive economy and inference. Cognitive economy means that , by viewing many things as instances of the same class, classification provides maximum information with the least cognitive effort. Inference means that identifying an instance as a member of a class makes it possible to draw conclusions. To decide upon potential classes two principles are discussed:

1. Abstraction form instances: A class can be defined only if there are instances in the relevant universe possessing all properties of the type that defines the class.
2. Maximal abstraction: A relevant property possessed by all instances of a class should be included in the class definition.

They propose two additional principles that apply to collections of classes: Completeness, which requires that all properties from the relevant universe be used in classification, and nonredundancy, which ensures that there are no redundant classes. A class that is a subclass of several other classes should be defined by at least one property not in any of its superclasses.

As indicated by [174], some works may use these terms somewhat differently:

- Some languages (like for instance SDM [160] and TAXIS [271]) represent aggregations by means of attributes (instead of cross product type construction). The part-of relation can be looked upon as a special case of aggregation. Based on work on object-oriented databases, this relation is further Specialized [264]. A set of component objects which form a single conceptual entity is referred to as a composite object, and the links connecting the components with this object are called part-of links. The model allows to specify for each composite link whether the reference is exclusive, i.e. the component exclusively belongs to the composite at a given point in time, or shared, meaning that the component may possibly be part-of several composites. Further, a part-of link can be defined to be either dependent, which means that the existence of the component depends on the existence of the composite, or independent, i.e. having existence irrespectively of the composite. These specializations are orthogonal, giving four possible relationships as exemplified below:
  1. A brain is part-of a person (exclusive, dependent).
  2. A paper is part-of a journal (exclusive, independent).
  3. A subprogram is part-of a program library (shared, dependent).
  4. A figure is part-of a paper (shared, independent).
- Some languages have identified several kinds of generalization. The following types of generalizations are defined by Kung [219].
  A set of *subclasses* of a *class* **cover** the *class* if all
  members of the *class* are *members* of at least one of the *subclasses*.
  A set of *subclasses* of a *class* are **disjoint** if no *members* of a *subclass* are *members* of any of the other *subclasses* of the *class*.
  A set of *subclasses* which are both *disjoint* and *cover* the *class* is called a **partition** of the *class*.
- It is often useful to define association in terms of the powerset operator. As suggested both by [174] and [301], association is commonly used for constructing sets of objects of the *same type*. Consider the example of Fig. 2.6 (taken from [174]), where the *-node denotes the association of the "person" node, meaning that the former is a subset of the powerset of the set of persons (i.e. each committee will have some group of members taken from the set of persons). Since we do not want to express at an abstract level the exact members of each committee, and since all members are persons, the association operator will have only one child in this case (whereas "men" and "women" being members of "sex-groups" earlier in this chapter signaled a use of association with several children).

2.1.3 Strengths and Weaknesses of Suggested Relations

We will here briefly discuss the strength and weaknesses of the suggested constructs.

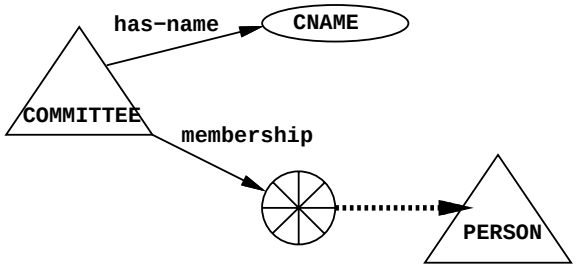


Fig. 2.6. Association with a single child

Strengths. As indicated by [174, 301, 307], many modeling languages provide at least some of the CAGA constructs, and the effects of introducing such constructs are positively described. [301] reports improvements in expressive economy, integrity maintenance, modeling flexibility, and modeling efficiency. But why is it that languages tend to predefine exactly CAGA and not any other hierarchical relations (like "is the boss of")? The main reason is their *generality* and *intuitivity*.

The generality of CAGA can be accounted for by the fact that they are asubstantial. Whereas relations like "is-the-father-of" and "is-the-boss-of" contain substantives "father" and "boss", whose semantics clearly limit the applicability of the relations, "is-part-of" uses the semantically very anonymous substantive "part". Anything can be a "part" of something – the set of potential fathers is much more limited. The substantives "instance", "subset", "member" are similarly weak in semantic content. Defined in terms of sets, with no commitment as to what these sets contain, these abstraction mechanisms should be able to cover any application area. Thus, they can be useful in organization modeling, process modeling, data modeling, hardware modeling etc.

Moreover, CAGA are apparently very intuitive abstraction mechanisms, which must be why they have become so popular in the first place. We seems to find it natural to think of things as being put together from smaller parts (aggregation), as being of a specific type (classification), as being members of groups (association) which can have smaller subgroups (generalization). This might partly be because we are being trained pretty much in using such hierarchies in school, for instance learning languages (aggregation: assembling words from letters, sentences from words, etc., classification: distinguishing between word classes, identifying phrases as subject, predicate, direct object, indirect object etc., generalization: different kinds of sentences, substantives, verbs etc., association: memorizing lists of prepositions demanding a certain case in German), learning biology, learning mathematics — whatever!

Weaknesses. However, there are also some weaknesses to be mentioned:

– The set-oriented definition of CAGA cause some limitations on their use.

– Also, there are hierarchies which are certainly of interest in conceptual modeling which are not covered by the CAGA scheme.

As can be seen from the set-theoretic definitions given in this chapter, classification means to move up one meta-level, from an instance to a type. The other three are set-level constructs. Thus, there are two problems:

– What to do about instances?
– What to do about masses?

1. Instances:. Instances are not necessarily such a big problem. The association construct is trivially applicable, since it can produce a set of instances just as easily as a set of sets ("Peter, Patricia, and Joey are members of the Party Committee"). Moreover, if we treat instances like sets with only one member (like Quine does in [309]), the definition of aggregation just presented is also trivially applicable ("The car # 346 was constructed from the chassis # 9213, the carossery # 2134, and the engine #905"), and so is generalization (with the limitation that it only seems to be useful in situations where the general notion is a variable: "Joey's murderer must have been either Peter or Patricia", in which case "Joey's murderer" can be said to be a generalization of "Peter" and "Patricia").

Another question is hierarchical relations between instances (like "father-of", "boss-of"). It is difficult to know which instance level relations people might want, and we cannot define an enormous amount of them in advance. The wisest thing for a general framework might be to provide a generic relation construct from which the users can define all the relations that they need.

2. Mass Concepts. As Sowa points out in [352] the set-oriented way of thinking which permeates so many information systems models work well for things that are countable, whereas there are problems for the so-called *mass nouns*, like water, love, money. Again it appears that the notions of AGA are applicable ("Chocolate is made of cocoa, sugar and milk" (aggregation), "Milk and water are both liquids" (generalization), "*Milk* and *Water* are members of the set *Liquids*" (association)). However, the problem is that we cannot use the set based definitions presented earlier in this chapter. There are two possible ways out of this:

– One can go for a more general definition of AGA which is not at all based on sets (but for instance on types).
– One can use the definitions already suggested and add some special tactics for dealing with masses.

We will not delve into this in more detail in this book.

2.2 Overview of Languages for Conceptual Modeling

In this section, we survey "the state of the art" of modeling languages, including those that have been applied in mature methodologies for system development and maintenance and some that are still on the research level. The overview will concentrate on the basic components and features of the languages to illustrate different ways of abstracting human perception of reality.

Modeling languages can be divided into classes according to the core phenomena classes that are represented in the language. We have called this the main *perspective* of the language. Another term often used, is *structuring principle*.

Generally, we can define a structuring principle to be *some rule or assumption concerning how data should be structured*. This is a very vague definition — we observe that

- A structuring principle can be more or less detailed: on a high level one for instance has the choice between structuring the information hierarchically, or in a general network. Most approaches take a far more detailed attitude towards structuring: deciding what is going to be decomposed, and how. For instance, structured analysis implies that the things to be decomposed are processes (maybe also stores and flows), and an additional suggestion might be that the hierarchy of processes should not be deeper than 4 levels, and the maximum number of processes in one decomposition 7.
- A structuring principle might be more or less rigid — in some approaches one can override the standard structuring principle if one wants to, in others this is impossible.

We will here basically discuss what we call *aggregation principles*. As stated in the previous section, aggregation means to build larger components of a system by assembling smaller ones. Going for a certain aggregation principle thus implies decision concerning

- What kind of components to aggregate.
- How other kinds of components (if any) will be connected to the hierarchical structure.

Fights between the supporters of different aggregation principles can often be rather heated. As we will show, the aggregation principle is a very important feature of an approach, so this is very understandable. Some possible aggregation principles are the following:

- Object-orientation.
- Process-orientation.
- Actor-orientation.

Objects are the things subject to processing, processes are the actions performed, and actors are the ones who perform the actions. Clearly, these three

approaches concentrate on different aspects of the perceived reality, but it is easy to be mistaken about the difference. It is not which aspects they capture and represent that are relevant. Instead, the difference is one of focus, representation, dedication, visualization, and sequence, in that an oriented language typically prescribes that [290]:

- Some aspects are promoted as fundamental for modeling, whereas other aspects are covered mainly to set the context of the promoted ones (focus).
- Some aspects are represented explicitly, others only implicitly (representation).
- some aspects are covered by dedicated modeling constructs, whereas others are less accurately covered by general ones (dedication).
- Some aspects are visualized in diagrams, others only recorded textually (visualization).
- Some aspects are captured before others during modeling(sequence).

Below we will investigate the characteristics of such perspectives in more detail.

2.2.1 An Overview of Modeling Perspectives

A traditional distinction regarding modeling perspectives is between the structural, functional, and behavioral perspective [283]. Yang [404], based on [235, 388], identifies a 'full' perspective to include the following:

- Data perspective. This is parallel to the structural perspective.
- Process perspectives. This is parallel to a functional perspective.
- Event/behavior perspective. The conditions by which the processes are invoked or triggered. This is covered by the behavioral perspective.
- Role perspectives. The roles of various actors carrying out the processes of a system.

In F3 [47], it is recognized that a requirement specification should answer the following questions:

- Why is the system built?
- Which are the processes to be supported by the system?
- Which are the actors of the organization performing the processes?
- What data or material are they processing or talking about?
- Which initial objectives and requirements can be stated regarding the system to be developed?

This indicate a need to support what we will term the *rule-perspective*, in addition to the other perspectives mentioned included by Yang.

In the NATURE project [186], one distinguishes between four worlds: Usage, subject, system, and development. Conceptual modeling as we use it here applies to the subject and usage world for which NATURE propose data

models, functional models, and behavior models, and organization models, business models, speech act models, and actor models respectively.

Based on the above, to give a broad overview of the different perspectives state-of-the-art conceptual modeling approaches accommodate, we have focused on the following perspectives:

- Structural perspective
- Functional perspective
- Behavioral perspective
- Rule perspective
- Object perspective
- Communication perspective
- Actor and role perspective

This is obviously only one way of classifying modeling approaches, and in many cases it will be difficult to classify a specific approach within this scheme. On the other hand, it is usuful way of ordering the presentation.

Another way of classifying modeling languages is according to their time-perspective [350]:

- Static perspective: Provide facilities for describing a snapshot of the perceived reality, thus only considering one state.
- Dynamic perspective: Provide facilities for modeling state transitions, considering two states, and how the transition between the states take place.
- Temporal perspective: Allow the specification of time dependant constraints. In general, sequences of states are explicitly considered.
- Full-time perspective: Emphasize the important role and particular treatment of time in modeling. The number of states explicitly considered at a time is infinite.

Another way of classifying languages are according to their level of formality. Conceptual modeling languages can be classified as semi-formal or formal, having a logical and/or executional semantics. They can in addition be used together with descriptions in informal languages and non-linguistic representations, such as audio and video recordings.

We will below present some languages within the main perspectives, and also indicate their temporal expressiveness and level of formality. Many of the languages presented here are often used together with other languages in so-called combined approaches. Some examples of such approaches will also be given.

2.2.2 The Structural Perspective

Approaches within the structural perspective concentrate on describing the static structure of a system. The main construct of such languages are the "entity". Other terms used for this role with some differences in semantics are object, concept, thing, and phenomena.

The structural perspective has traditionally been handled by languages for data modeling. Whereas the first data modeling language was published in 1974 [174], the first having major impact was the *entity-relationship* language of Chen [62].

Basic Vocabulary and Grammar of the ER-language:. In [62], the basic components are:

- **Entities**. An *entity* is a phenomenon that can be distinctly identified. Entities can be classified into entity classes;
- **Relationships**. A *relationship* is an association among entities. Relationships can be classified into relationship classes;
- **Attributes and data values**. A value is used to give value to a property of an entity or relationship. Values are grouped into value classes by their types. An *attribute* is a function which maps from an entity class or relationship class to a value class; thus the property of an entity or a relationship can be expressed by an attribute-value pair.

An ER-model contains a set of entity classes, relationship classes, and attributes. An example of a simple ER-model is given in Fig. 3.3.

Several extensions have later been proposed for so-called semantic data modeling languages [174, 301], with specific focus on the addition of mechanisms for hierarchical abstraction.

Basic Vocabulary and Grammar for Semantic Data Modeling Language:. In Hull and King's overview [174] a generic semantic modeling language (GSM) is presented. Figure 2.7 illustrates the vocabulary of GSM:

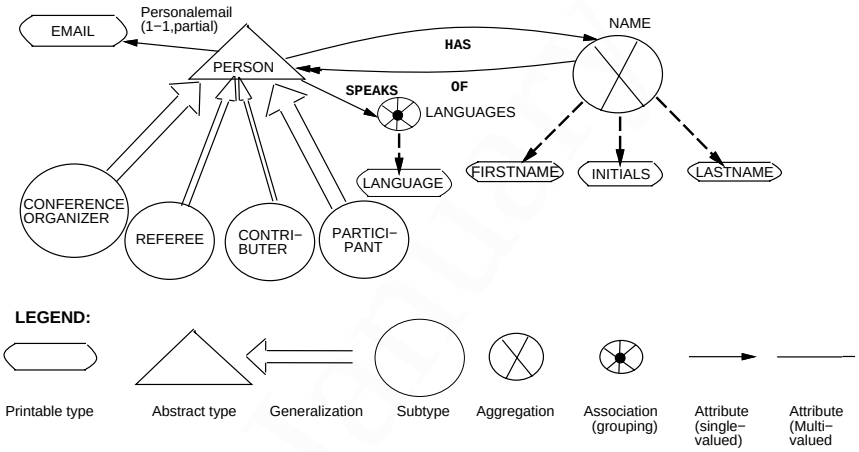


Fig. 2.7. Example of a GSM model

- **Primitive types**. The data types in GSM are classified into two kinds: the printable data types, that are used to specify some visible values, and the abstract types that represent some entities. In the example, the following printable types can be identified: *Email-address, language, firstname, initials,* and *lastname*.
- **Constructed types built by means of abstraction**. The most often used constructors for building abstractions are generalization, aggregation, and association. In the example we find *Person* as an abstract type, with specializations *conference organizer, referee, contributer,* and *participant*. *Name* is an aggregation of *firstname, initials,* and *lastname*, whereas *languages* is an association of a set of *language*
- **Attributes**.

In addition it is possible to specify derived classes in GSM.

Relationships between instances of types may be defined in different ways. We see in Fig. 2.7 that a relationship is defined by a two-way attribute (an attribute and its inverse). In the ER modeling language, a relationship is represented as an explicit type. The definition of relationship types provides the possibility of specifying such relationships among the instances of more than two types as well as that of defining attributes of such relationship types.

Other approaches:. The NIAM language [273] is a binary relationship language, which means that relationships that involve three or more entities are not allowed. Relationships with more than two involved parts will thus have to be objectified (i.e. modeled as entity sets instead). In other respects, the NIAM language has many similarities with ER, although often being classified as a form of object-role modeling. The distinction between entities and printable values is reflected in NIAM through the concepts of lexical and non-lexical object types, where the former denote printable values and the latter abstract entities. Aggregation is provided by the relationship construct just like in ER, but NIAM also provides generalization through the subobject-type construct. The diagrammatic notation is rather different from ER, but we will not discuss the details of this here. Another binary relationship language, ERT, will be briefly presented as part of the presentation on Tempora in Sect. 2.2.5. A distinguishing feature of this language is the modeling of *temporal* aspects.

Another type of structural modeling languages are semantic networks [350]. A semantic network is a graph where the nodes are objects, situations, or lower level semantic networks, and the edges are binary relations between the nodes. Semantic networks constitute a large family of languages with very diverse expressive power. Sowa's conceptual graph formalism [352] can be said to be a special kind of semantic network language. The language is based on work within linguistics, psychology, and philosophy. In the models, concept nodes represent entities, attributes, states, and events, and relation nodes show how the phenomena classes are interconnected. A conceptual

graph is a finite, connected, bipartite graph. Every conceptual relation has one or more arcs.

Each conceptual graph asserts a single proposition and has no meaning in isolation. Only through a semantic network are its concepts and relations linked to context, language, emotion, and perception. Figure 2.8 shows a conceptual graph for the proposition *a cat sitting on a mat*. Dotted lines link the nodes of the graph to other parts of the semantic network.

- Concrete concepts are associated with percepts for experiencing the world and motor mechanisms for acting upon it.
- Some concepts are associated with the vocabulary and grammar rules of a language.
- A hierarchy of concept types defines generalization relationships between concepts.
- Formation rules determine how each type of concept may be linked to conceptual relations.
- Each conceptual graph is linked to some context or episode to which it is relevant.
- Each episode may also have emotional associations, which indirectly confer emotional overtones on the types of concepts involved.

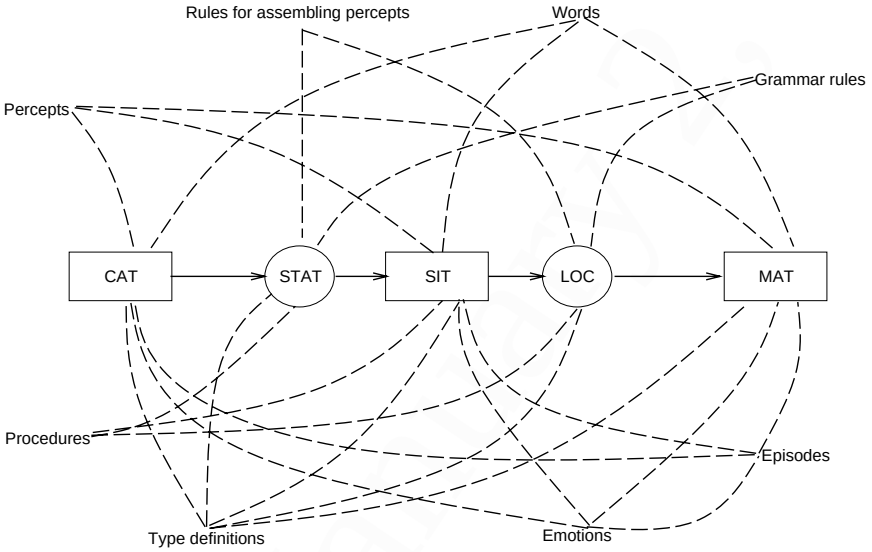


Fig. 2.8. A conceptual graph linked to a semantic network (From [352])

Also many object-oriented modeling languages can be classified as having a structure perspective. Object-orientation is discussed further in Sect. 2.2.6 and Sect. 2.2.8.

2.2.3 The Functional Perspective

The main phenomena class in the functional perspective is the process: A process is defined as an activity which based on a set of *phenomena* transforms them to a possibly empty set of *phenomena*.

The best know conceptual modeling language with a process perspective is data flow diagrams (DFD) [129] which describes a situation using the symbols illustrated in Fig. 2.9:

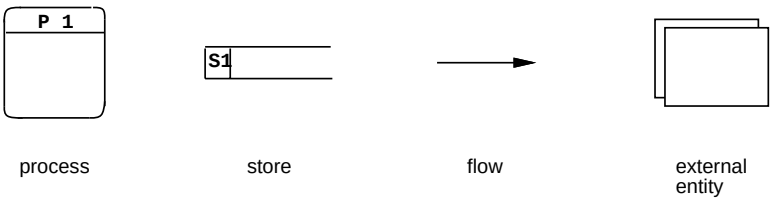


Fig. 2.9. Symbols in the DFD language

- **Process**. Illustrates a part of a system that transforms a set of inputs to a set of output:
- **Store**. A collection of data or material.
- **Flow**. A movement of data or material within the system, from one system component (process, store, or external entity) to another;
- **External entity**. An individual or organizational actor, or a technical actor that is outside the boundaries of the system to be modeled, which interact with the system.

With these symbols, a system can be represented as a network of processes, stores and external entities linked by flows. A process can be decomposed into a new DFD. When the description of the process is considered to have reached a detailed level where no further decomposition is needed, "process logic" can be defined in forms of e.g. structured English, decision tables, and decision trees.

When a process is decomposed into a set of sub-processes, the sub-processes are grouped around the higher level process, and are co-operating to fulfill the higher-level function. This view on DFDs has resulted in the "context diagram" [129] that regards the whole system as a process which receives and sends all inputs and outputs to and from the system. A context diagram determine the boundary of a system. Every activity of the system is seen as the result of a stimulus by the arrival of a data flow across some boundary. If no external data flow arrives, then the system will remain in a stable state. Therefore, a DFD is basically able to model reactive systems.

DFD is a semi-formal language. Some of the short-comings of DFD regarding formality are addressed in the transformation schema presented by Ward [390]. The main symbols of his language are illustrated in Fig. 2.10.

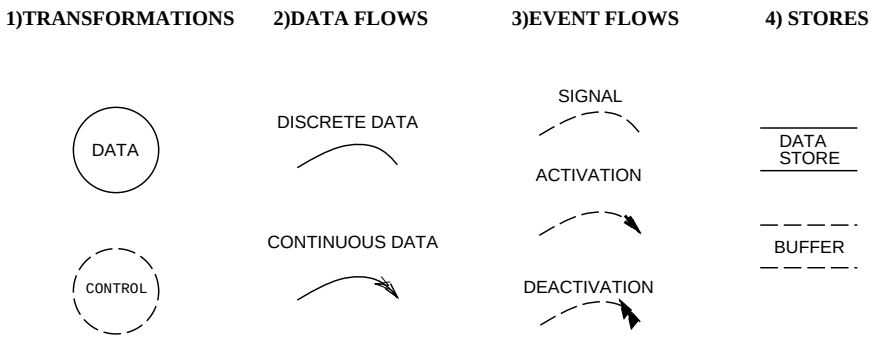


Fig. 2.10. Symbols in the transformation schema language

There are four main classes of symbols:

- **1. Transformations**: A solid circle represent a data transformation, which are used approximately as a process in DFD. A dotted circle represents a control transformation which controls the behavior of data transformations by activating or deactivating them, thus being an abstraction on some portion of the systems' control logic.
- **2. Data flows**: A discrete data flow is associated with a set of variable values that is defined at discrete points in time. Continuous data flows are associated with a value or a set of values defined continuously over a time-interval.
- **3. Event flows**: These report a happening or give a command at a discrete point in time. A signal shows the sender's intention to report that something has happened, and the absence of any knowledge on the sender's part of the use to which the signal is put. Activations show the senders intention to cause a receiver to produce some output. A deactivation show the senders intention to prevent a receiver from producing some output.
- **4. Stores**: A store acts as a repository for data that is subject to a a storage delay. A buffer is a special kind of store in which flows produced by one or more transformations are subject to a delay before being consumed by one or more transformations. It is an abstraction of a stack or a queue.

Both process and flow decomposition are supported.

Whereas Ward had a goal of formalizing DFD's , Opdahl and Sindre [287, 289] try to adapt data flow diagrams to what they term 'real-world modeling'.

Problems they note with DFD in this respect are as follows:

- 'Flows' are semantically overloaded: Sometimes a flow means transportation, other times it merely connects the output of one process to the input of the next.
- Parallelism often has to be modeled by duplicating data on several flows. This is all right for data, but material cannot be duplicated in the same way.

– Whereas processes can be decomposed to contain flows and stores in addition to sub-processes, decomposition of flows and stores is not allowed. This makes it hard to deal sensibly with flows at high levels of abstraction [46].

These problems have been addressed by unifying the traditional DFD vocabulary with a taxonomy of real-world activity, shown in Table 2.1: The three DFD phenomena "process," "flow", and "store" correspond to the physical activities of "transformation," "transportation", and "preservation" respectively. Furthermore, these three activities correspond to the three fundamental aspects of our perception of the physical world: matter, location, and time. Hence, e.g., an *ideal flow* changes the location of items in zero time and without modifying them.

Since these ideal phenomena classes are too restricted for high level modeling, real phenomena classes were introduced. Real processes, flows, and stores are actually one and the same, since they all can change all three physical aspects, i.e., these are fully inter-decomposable. The difference is only subjective, i.e., a real-world process is mainly perceived as a transformation activity, although it may also use time and move the items being processed.

Additionally, the problem with the overloading of 'flow' is addressed by introducing a *link*, for cases where there are no transportation. Links go between *ports* located on various processes, stores and flows, and may be associated with spatial coordinates

[287] also provides some definitions relating to the *items* to be processed, including proper distinctions between data and material. Items have *attributes* which represent the properties of data and materials, and they belong to *item classes.* Furthermore classes are related by the conventional abstraction relations aggregation, generalization, and association. Hence the specification of item classes constitute a *static* model which complements the dynamic models comprising processes, flows, stores, and links.

Table 2.1. A data flow diagram taxonomy of real-world dynamics

Phenomena class	Process	Flow	Store
Activity	Transformation	Transportation	Preservation
Aspect	Matter	Location	Time

The symbols in the language are shown in Fig. 2.11. The traditional DFD notation for processes and flows are retained, however, to facilitate the visualization of decomposition, it is also possible to depict the flow as an enlarged kind of box-arrow. Similarly, to facilitate the illustration of decomposed stores, full rectangles instead of open-ended ones are used. Links are shown as dotted arrows.

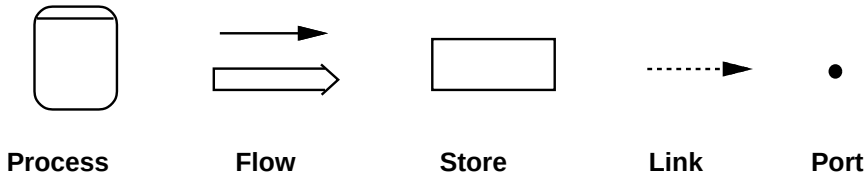


Fig. 2.11. Symbols in the real-world modeling language

2.2.4 The Behavioral Perspective

In most languages with a behavioral perspective the main phenomena are states and transitions between states. State transitions are triggered by events [79].

A finite state machine (FSM) is a hypothetical machine that can be in only one of a given number of states at any specific time. In response to an input, the machine generates an output, and changes state. There are two language-types commonly used to model FSM's: State transition diagrams (STD) and state transition matrices (STM). The vocabulary of state transition diagrams is illustrated in Fig. 2.12 and are described below:

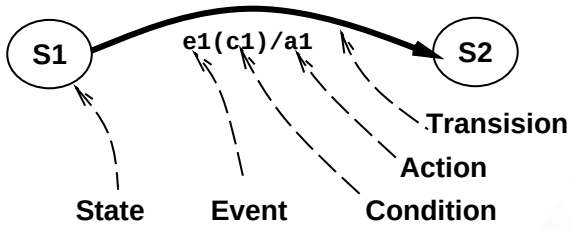


Fig. 2.12. Symbols in the state transition modeling language

- **State**: A system is always in one of the states in the lawful state space for the system. A state is defined by the set of transitions leading to that state, the set of transitions leading out of that state and the set of values assigned to attributes of the system while the system resides in that state.
- **Event**: An event is a message from the environment or from system itself to the system. The system can react to a set of predefined events.
- **Condition**: A condition for reacting to an event. Another term used for this is 'guard'.
- **Action**: The system can perform an action in response to an event in addition to the transition.
- **Transition**: Receiving an event will cause a transition to a new state if the event is defined for the current state, and if the condition assigned to the event evaluates to true.

A simple example that models the state of a paper during the preparation of a professional conference is depicted in Fig. 2.13. The double circles indicate end-states.

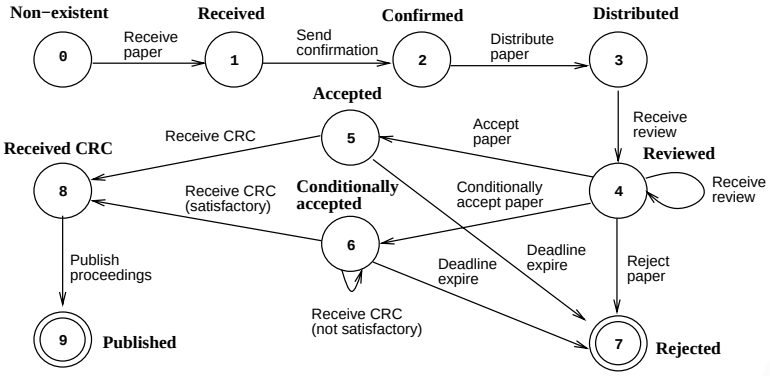


Fig. 2.13. Example of a state transition model

In a STM a table is drawn with all the possible states labeling the rows and all possible stimuli labeling the columns. The next state and the required system response appear at each intersection [80]. In basic finite state machine one assume that the system response is a function of the transition. This is the Mealy model of a finite state machine. An alternative is the Moore model in which system responses are associated with the state rather than the transitions between states. Moore and Mealy machines are identical with respect to their expressiveness.

It is generally acknowledged that a complex system cannot be beneficially described in the above fashion, because of the unmanageable, exponentially growing multitude of states, all of which have to be arranged in a 'flat' model. Hierarchical abstraction mechanisms are added to traditional STD in Statecharts [161] to provide the language with modularity and hierarchical construct as illustrated in Fig. 2.14.

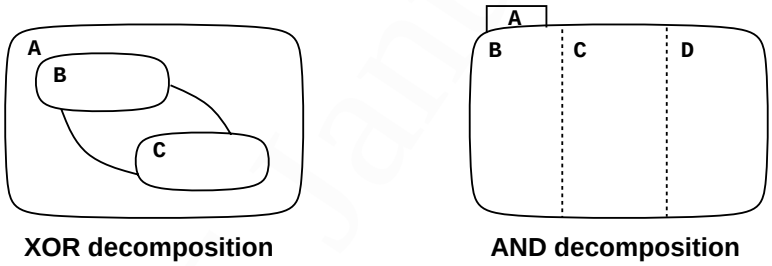


Fig. 2.14. Decomposition mechanisms in Statecharts

- **XOR decomposition**: A state is decomposed into several states. An event entering this state (A) will have to enter one and only one of its sub-states (B or C). In this way generalization is supported.
- **AND decomposition**: A state is divided into several states. The system resides in *all* these states (B, C, and D) when entering the decomposed state (A). In this way aggregation is supported.

One has introduced the following mechanisms to be used with these abstractions:

- **History**: When entering the history of a XOR decomposed state, the sub-state which was visited last will be chosen.
- **Deep History**: The semantics of history repeated all the way down the hierarchy of XOR decomposed states.
- **Condition**: When entering a condition inside a XOR decomposed state, one of the sub-states will be chosen to be activated depending on the value of the condition.
- **Selection**: When entering a selection in a state, the sub-state selected by the user will be activated.

In addition support for the modeling of delays and time-outs is included.

Fig. 2.15 shows the semantics behind these concepts and various activating methods available.

Statecharts are integrated with functional modeling in [164]. Later extensions of statecharts for object-oriented modeling is reported in [68, 163, 319]. The latter of these will be described in Sect. 2.2.6.

Petri-Nets. Petri-nets [304] is another well-known behaviorally oriented modeling language. A model in the original Petri-net language is shown in Fig. 2.16. Here, *places* indicate a system state space, and a combination of *tokens* included in the places determine the specific system state. State *transitions* are regulated by firing rules: A transition is enabled if each of its input places contains a token. A transition can fire at any time after it is enabled. The transition takes zero time. After the firing of a transition, a token is removed from each of its input places and a token is produced in all output places.

Figure 2.16 shows how dynamic properties like precedence, concurrency, synchronization, exclusiveness, and iteration can be modeled in a Petri-net. The associated model patterns along with the firing rule above establish the execution semantics of a Petri-net.

The classical Petri net cannot be decomposed. This is inevitable by the fact that transitions are instantaneous, which makes it impossible to compose more complex networks (whose execution is bound to take time) into higher level transitions. However, there exists several more recent dialects of the Petri net language (for instance [253])) where the transitions are allowed to take time, and these approaches provide decomposition in a way not very

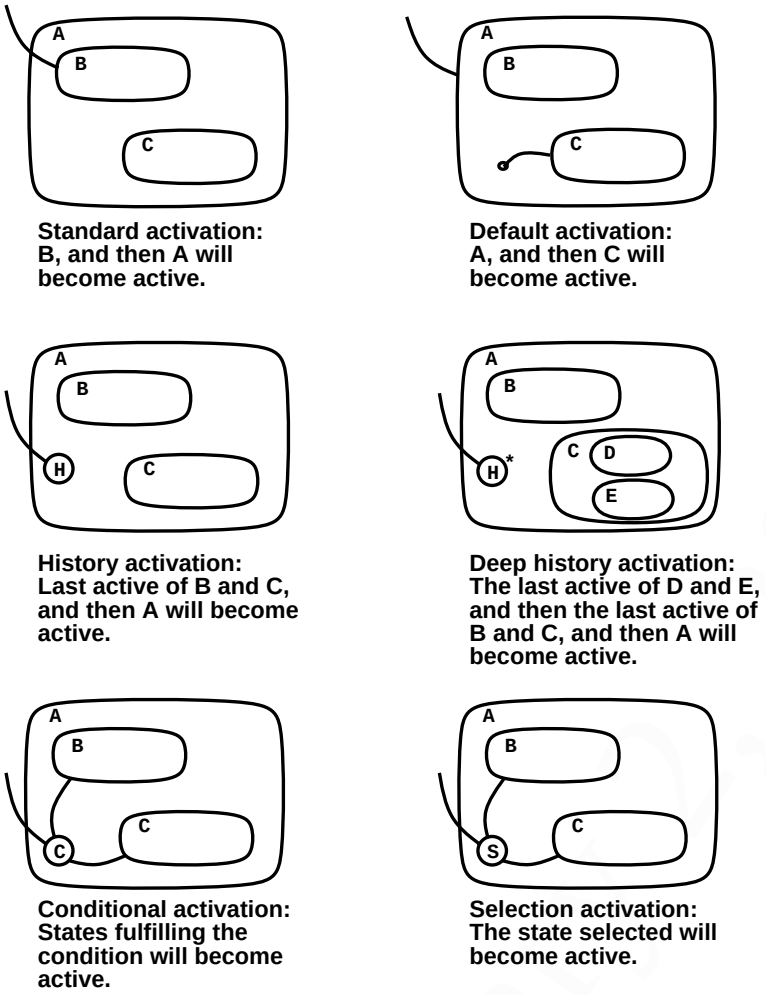


Fig. 2.15. Activation mechanisms in Statecharts

different from that of a data flow diagram. Timed Petri Nets [253] also provide probability distributions that can be assigned to the time consumption of each transition and is particularly suited to performance modeling.

BNM (Behavior Network Model) is a language for describing information system structure and behavior — an example diagram is shown in Fig. 5.4. The language uses Sølvbergs Phenomenon Model [348] for data modeling, coupled with an extended Petri net formalism for dynamic modeling. This coupling is shown by edges between places in the Petri net and phenomenon classes. The token of a place can either be an element of a phenomenon class

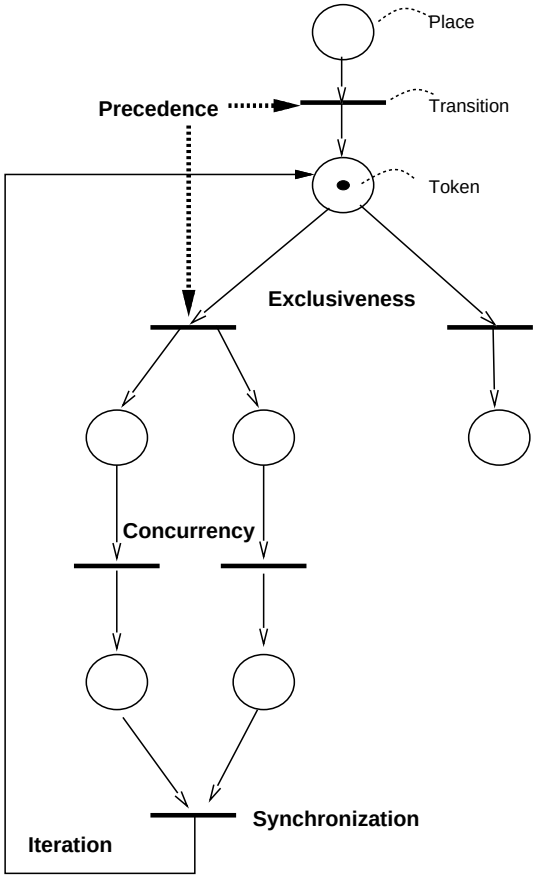


Fig. 2.16. Dynamic expressiveness of Petri-nets

(the edge is annotated with "$\in$", e.g. *Messages* in the example) or it can be the whole class (the edge is annotated with "=", e.g. *Orders* in the example).

The Petri nets of BNM differ from standard Petri nets in that

- Tokens are named and typed variables, i.e. one have a so-called *colored* Petri-net. Class variables have capital letters and element variables have small letters.
- There are two kinds of input places to a transition: consumption places and reference places. For the former, a token is consumed when a transition fires, whereas the latter is not consumed. A reference place is indicated by a dotted line.
- Transitions are allowed to take time.
- Transitions have pre- and postconditions in predicate logic. For a transition to fire, its precondition must be true, and by the firing its postcondition will become true.

Otherwise, the BNM semantics are in accordance with standard Petri net semantics.

2.2.5 The Rule Perspective

A rule has been defined as follows:

> A **rule** is something which influences the *actions* of a non-empty set of *actors*. A rule is either a rule of necessity or a deontic rule [393].
> A **rule of necessity** is a *rule* that must always be satisfied. It is either analytic or empirical (see below).
> A *rule of necessity* which can not be broken because of an inter-subjectively agreed definition of the terms used in the rules is called **analytic**.
> A *rule of necessity* that can not be broken according to present *shared explicit knowledge* is called **empirical**.
> A **deontic rule** is a *rule* which is only socially agreed among a set of persons. A deontic rule can thus be violated without redefining the terms in the rule. A deontic rule can be classified as being an obligation, a recommendation, a permission, a discouragement, or a prohibition [214].

The general structure of a rule is

" if *condition* then *expression*"

where *condition* is descriptive, indicating the scope of the rule by designating the conditions in which the rule apply, and the *expression* is prescriptive. According to Twining [373] any rule, however expressed, can be analyzed and restated as a compound conditional statement of this form.

Current Applications. Representing knowledge by means of rules is not a novel idea. According to Davis and King [82], production systems were first proposed as a general computational mechanism by Post in 1943. Today, rules are used for knowledge representation in a wide variety of applications, such as expert systems, tutoring and planning systems, database systems and requirement specification in general. It is the use of rules within requirement specification that will be our focus here.

Several advantages have been experienced with a declarative, rule-based approach to information systems modeling:

- *Problem-orientation:* The representation of business rules declaratively is independent of what they are used for and how they will be implemented. With an explicit specification of assumptions, rules, and constraints, the analyst has freedom from technical considerations to reason about application problems [85, 153]. This freedom is even more important for the communication with the stakeholders with a non-technical background [37, 45, 155, 374].

- *Maintenance:* A declarative approach makes possible a *one place representation* of every rule and fact, which is a great advantage when it comes to the maintainability of the specification [281].
- *Knowledge enhancement:* The rules used in an organization, and as such in a supporting CIS, are not always explicitly given. In the words of Stamper [354] "Every organization, in as far as it is organized, acts as though its members were confronting to a set of rules only a *few of which may be explicit* [4]." This has inspired certain researchers to look upon CIS specification as a process of rule reconstruction [143], i.e. the goal is not only to represent and support rules that are already known, but also to uncover de facto and implicit rules which are not yet part of a shared organizational reality, in addition to the construction of new, possibly more appropriate ones.

On the other hand, several problems have been observed when using a simple rule-format. Although addressed in different ways in different areas, many of these also applies to the use of rules for conceptual modeling.

- Every statement must be either true or false, there is nothing in between.
- It is usually not possible to distinguish between rules of necessity and deontic rules [395].
- In many rule modeling languages it is not possible to specify who the rules apply to.
- Formal rule languages have the advantage of eliminating ambiguity. However, this does not mean that rule based models are easy to understand. There are two problems with the comprehension of such models, both the comprehension of single rules, and the comprehension of the whole rule-base. Whereas the traditional operational models have decomposition and modularization facilities which make it possible to view a system at various levels of abstraction and to navigate in a hierarchical structure, rule models are usually *flat*. With many rules such a model soon becomes difficult to grasp, even if each rule should be understandable in itself. According to Li [233] this often makes rule-based systems both unmaintainable and untestable and as such unreliable.
- A general problem is that a set of rules is either consistent or inconsistent. On the other hand, human organizations may often have more or less contradictory rules.

Some approaches to rule-based modeling that tries to address some of these problems are presented below.

COMEX [383, 384] is a tool for editing and executing task models. The task model is based on PPM in PPP (see description in Sect. 2.5). A task corresponds to a process in a DFD. Each task is associated with a set of rules and has a coupling between the task model and the rules similar to the one in Tempora.

[4] Our italics.

Tempora . Tempora [243] was an ESPRIT-3 project that finished in 1994. It aimed at creating an environment for the development of complex application systems. The underlying idea was that development of a CIS should be viewed as the task of developing the rule-base of an organization, which is used throughout development.

Tempora has three closely interrelated languages for conceptual modeling. ERT [256, 367], being an extension of the ER language, PID [152, 367], being an extension of the DFD in the SA/RT-tradition, and ERL [254, 367], a formal language for expressing the rules of an organization.

The ERT Language. The basic modeling constructs of ERT are: Entity classes, relationship classes, and value classes. The language also contains the most usual constructs from semantic data modeling [301] such as generalization and aggregation, and derived entities and relationships, as well as some extensions for temporal aspects particular for ERT. It also has a grouping mechanism to enhance the visual abstraction possibilities of ERT models. The graphical symbols of ERT are Shown in Fig. 2.17.

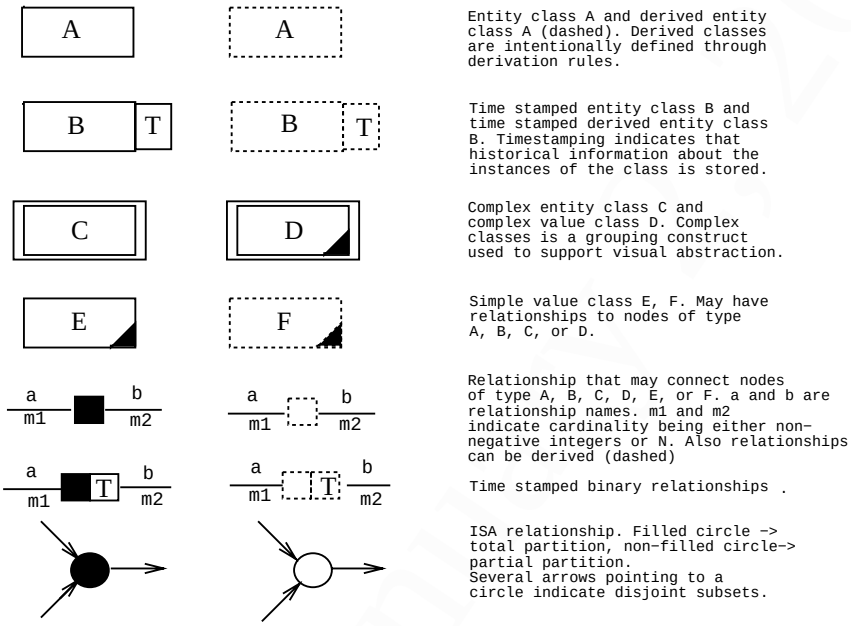


Fig. 2.17. Symbols in the ERT languages

The PID Language. This language is used to specify processes and their interaction in a formal way. The basic modeling constructs are: processes, ERT-views being links to an ERT-model, external agents, flows (both control and data flows), ports, and timers, acting as either clocks or delays. The graphical symbols of PID's are shown in Fig. 2.18.

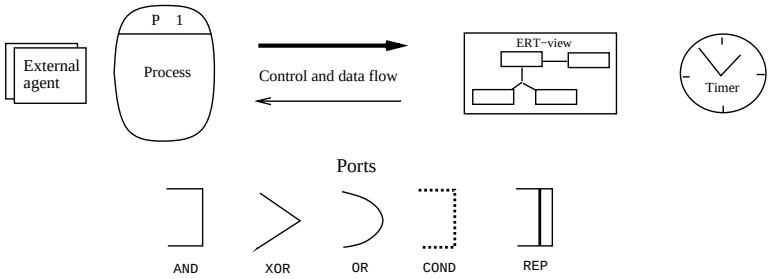


Fig. 2.18. Symbols in the PID language

The External Rule Language (ERL). The ERL is based on first-order temporal logic, with the addition of syntax for querying the ERT model. The general structure of an ERL rule is as follows:

when *trigger* if *condition*, then *consequence* else *consequence*.

- *trigger* is optional. It refers to a state change, i.e. the rule will only be enabled in cases where the trigger part becomes true, after having been previously false. The trigger is expressed in a limited form of first order temporal logic.
- *condition* is an optional condition in first order temporal logic.
- *consequence* is an action or state which should hold given the trigger and condition. The consequence is expressed in a limited form of first order temporal logic. The 'else' clause indicates the *consequence* when the condition is not true, given the same trigger.

ERL-rules have both declarative and procedural semantics. To give procedural semantics to an ERL-rule, it must be categorized as being a *constraint*, a *derivation rule*, or an *action rule*. In addition, it is possible to define *predicates* to simplify complex rules by splitting them up into several rules.

The rule can be expressed on several levels of details from a natural language form to rules which can be executed.

- *Constraints* express conditions on the ERT database which must not be violated.
- *Derivation rules* express how data can be automatically derived from data that already exist.
- *Action rules* express which actions to perform under what conditions. Action rules are typically linked to atomic processes in the process model, giving the execution semantics for the processes as illustrated in Fig. 2.19. A detailed treatment of the relationship between processed and rules is given in [255, 331].

The main extension in ERL compared to other rule-languages is the temporal expressiveness. At any time during execution, the temporal database

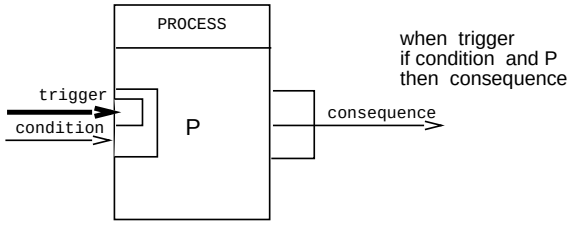


Fig. 2.19. Relationship between the PID and ERL languages (from [212])

will have stored facts not only about the present time, but also about the past and the future. This is viewed as a sequence of databases, each associated with some *tick*, and one may query any of these databases. ERL rules are always evaluated with respect to the database that corresponds to the real time the query is posed.

In addition to linking PID to ERT-models and ERL-rules to ERT-models and PIDs, one have the possibility of relating rules in rule hierarchies. The relationships available for this in Tempora are [330, 344]:

- *Refers-to:* Used to link rules where definitions or the introduction of a necessary situation can be found in another rule.
- *Necessitates* and *motivates*: Used to create goal-hierarchies.
- *Overrules* and *suspends*: These deal with exceptions. If an action is overruled by another rule, then it will not be performed at all, whereas an action which is suspended, can be performed when the condition of the suspending rule no longer holds. With these two relations, exceptions can be stated separately and then be connected to the rules they apply to. This provides a facility for hiding details, while obtaining the necessary exceptional behavior when it is needed.

Tempora is one of many *goal-oriented approaches* that has appeared in the nineties, Other such approaches are described below. In the ABC method developed by SISU [397] a goal-model is supported, where goals can be said to obstruct, contribute to, or imply other goals. A similar model is part of the F3 modeling languages [47]. Other examples of goal-oriented requirement approaches is reported by Feather [114] where the possible relations between goals and policies are *Supports*, *Impedes*, and *Augments*. Goals can also be *subgoals* i.e decompositions of other goals. Sutcliffe and Maiden [356], and Mylopoulos et al. [270] who use a rule-hierarchy for the representation of non-functional requirements are other examples which we will describe further below.

Sutcliffe. [356] differentiate between six classes of goals:

- 1. Positive state goals: Indicate states which must be achieved.
- 2. Negative state goals: Express a state to be avoided.

- 3. Alternative state goal: The choice of which state applies depends on input during run-time.
- 4. Exception repair goal: In these cases nothing can be done about the state an object achieves, even if it is unsatisfactory and therefore must be corrected in some way.
- 5. Feedback goals: These are associated with a desired state and a range of exceptions that can be tolerated.
- 6. Mixed state goals: A mixture of several of the above.

For each goal-type there is defined heuristics to help refine the different goal-types. Most parent nodes in the hierarchy will have 'and' relations with the child nodes, as two or more sub-goals will support the achievement of a higher level goal, however there may be occasions when 'or' relations are required for alternatives. Goals are divided into policies, functional goals and domain goals. The policy level describes statements of what should be done. The functionally level has linguistic expressions containing some information about how the policy might be achieved. Further relationship types may be added to show goal conflicts, such as 'inhibits', 'promotes', and 'enables' to create an argumentation structure. On the domain level templates are used to encourage addition of facts linking the functional view of aims and purpose to a model in terms of objects, agents, and processes.

Figure 2.20 illustrates a possible goal hierarchy for a library indicating examples of the different goal-types.

Mylopoulos et al. [63, 270] describes a similar language for representing non-functional requirements, e.g. requirements for efficiency, integrity, reliability, usability, maintainability, and portability of a CIS. The framework consists of five major components:

1. A set of goals for representing non-functional requirements, design decisions and arguments in support of or against other goals.
2. A set of link types for relating goals and goal relationships.
3. A set of generic methods for refining goals into other goals.
4. A collection of correlation rules for inferring potential interaction among goals.
5. A labeling procedure which determines the degree to which any given non-functional requirement is being addressed by a set of design decisions.

Goals are organized into a graph-structure in the spirit of and/or-trees, where goals are stated in the nodes. The goal structure represents design steps, alternatives, and decisions with respect to non-functional requirements. Goals are of three classes:

- Nonfunctional requirements goals: This includes requirements for accuracy, security, development, operating and hardware costs, and performance.
- Satisficing goals: Design decisions that might be adopted in order to satisfice one or more nonfunctional requirement goal.

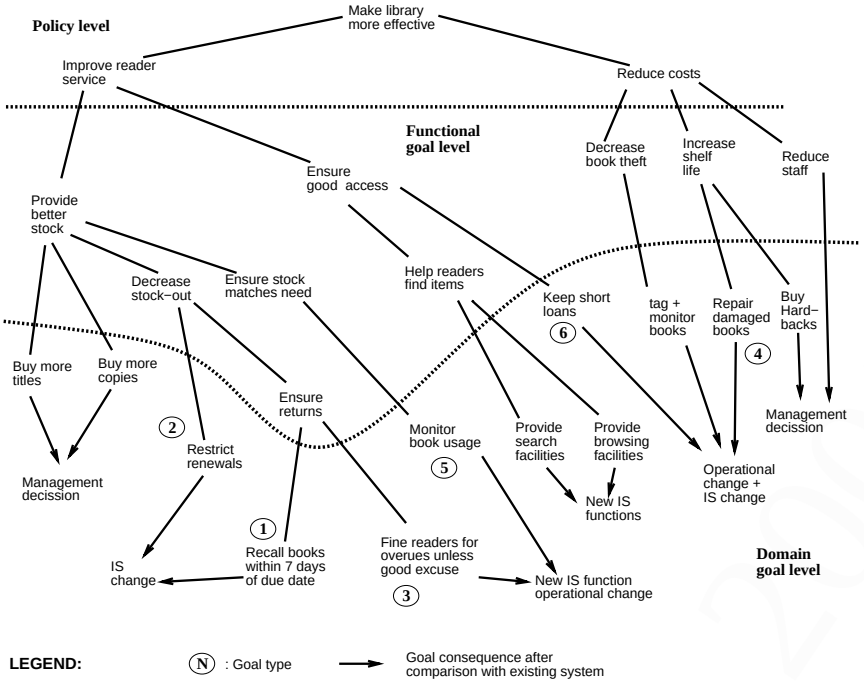


Fig. 2.20. Example of a goal hierarchy (From [356])

– Arguments: Represent formally or informally stated evidence or counter-evidence for other goals or goal-refinements.

Nodes are labeled as undetermined (U), satisficed (S) and denied (D).

The following link types are supported describing how the satisficing of the offspring or failure thereof relates to the satisficing of the parent goal:

– *sub*: The satisficing of the offspring contributes to the satisficing of the parent.
– *sup*: The satisficing of the offspring is a sufficient evidence for the satisficing of the parent.
– *-sub*: The satisficing of the offspring contributes to the denial of the parent
– *-sup*: The satisficing of the offspring is a sufficient evidence for the denial of the parent.
– *und*: There is a link between the goal and the offspring, but the effect is as yet undetermined.

Links can relate goals, but also links between links and arguments are possible. Links can be induced by a method or by a correlation rule (see below).

Goals may be refined by the modeler, who is then responsible for satisficing not only the goal's offspring, but also the refinement itself represented as a link. Alternatively, the framework provides goal refinement methods which

represent generic procedures for refining a goal into one or more offsprings. These are of different kinds: Goal decomposition methods, goal satisficing methods, and argumentation methods.

As indicated above, the non-functional requirements set down for a particular system may be contradictory. Guidance is needed in discovering such implicit relationship and in selecting the satisficing goals that best meet the need of the non-functional goals. This is achieved either through external input by the designer or through generic correlation rules.

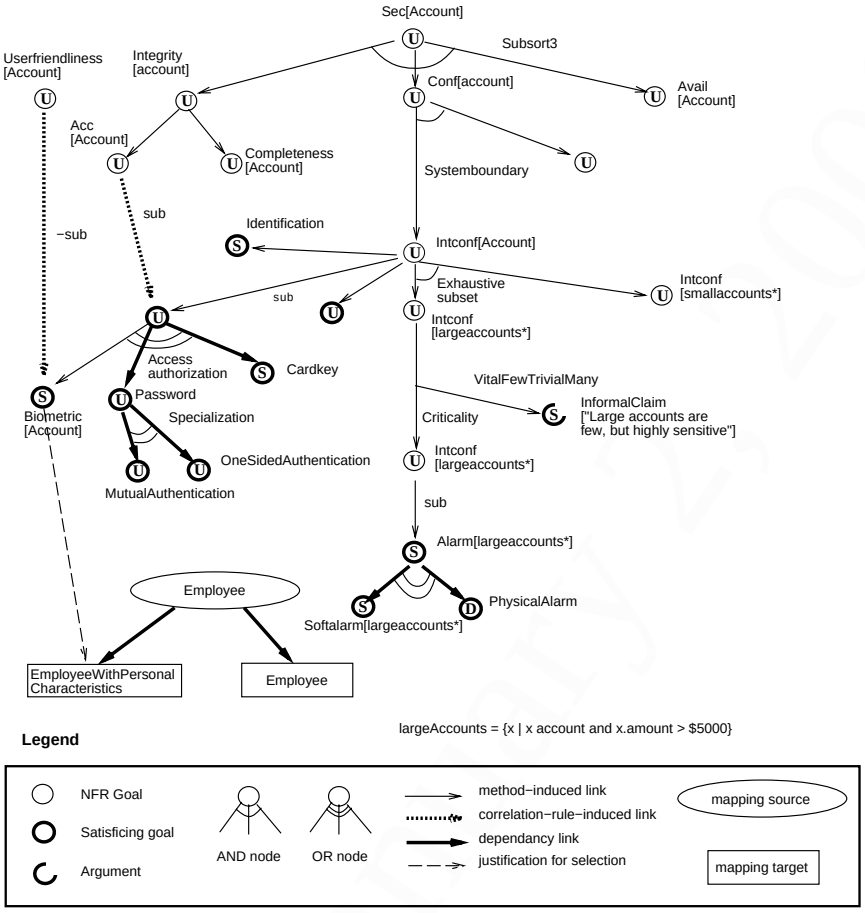


Fig. 2.21. Example of a goal-graph (From [63])

An example showing how to fulfill the security requirements of a bank's credit card system is given in Fig. 2.21. The example shows how to fulfill the security requirements of a bank's credit card system. Starting from the top, the method Subsort3 is used to decompose the goal into three other goals for

integrity, confidentiality and availability. A correlation rule comes into play when an offspring has an impact on some goals other than the parent.

2.2.6 The Object Perspective

The basic phenomena of object oriented modeling languages are similar to those found in most object oriented programming languages:

- **Object**: An *object* is an "entity" which has a unique and unchangeable identifier and a local state consisting of a collection of attributes with assignable values. The state can only be manipulated with a set of *methods* defined on the object. The value of the state can only be accessed by sending a *message* to the object to call on one of its methods. The details of the methods may not be known, except through their interfaces. The happening of an operation being triggered by receiving a message, is called an *event*.
- **Process**: The *process* of an object, also called the object's *life cycle*, is the trace of the events during the existence of the object.
- **Class**: A set of objects that share the same definitions of attributes and operations compose an *object class*. A subset of a class, called *subclass*, may have its special attribute and operation definitions, but still share all definitions of its superclass through *inheritance*.

A survey of current object-oriented modeling approaches is given in [396]. According to this, object-oriented analysis should provide several representations of a system to fully specify it:

- Class relationship models: These are similar to ER models.
- Class inheritance models: Similar to generalization hierarchies in semantic data-models.
- Object interaction models: Show message passing between objects
- Object state tables (or models): Follow a state-transition idea as found in the behavioral perspective.
- User access diagrams: User interface specification.

A general overview of phenomena represented in object-modeling languages is given in Fig. 2.22.

These break down into structural, behavioral, and rules, cf. Sect. 2.2.2,Sect. 2.2.4, and Sect. 2.2.5.

Static phenomena break down into type-related and class-related. A *type* represents a definition of some set of phenomena with similar behavior. A *class* is a description of a group of phenomena with similar properties. A class represents a particular implementation of a type. The same hierarchical abstraction mechanisms found in semantic data models are also found here. *Inheritance* is indicated as a generalization of the generalization-mechanism. Classes or types bound by this kind of relationship share attributes and operations. Inheritance can be either *single* – where a class or type can have no

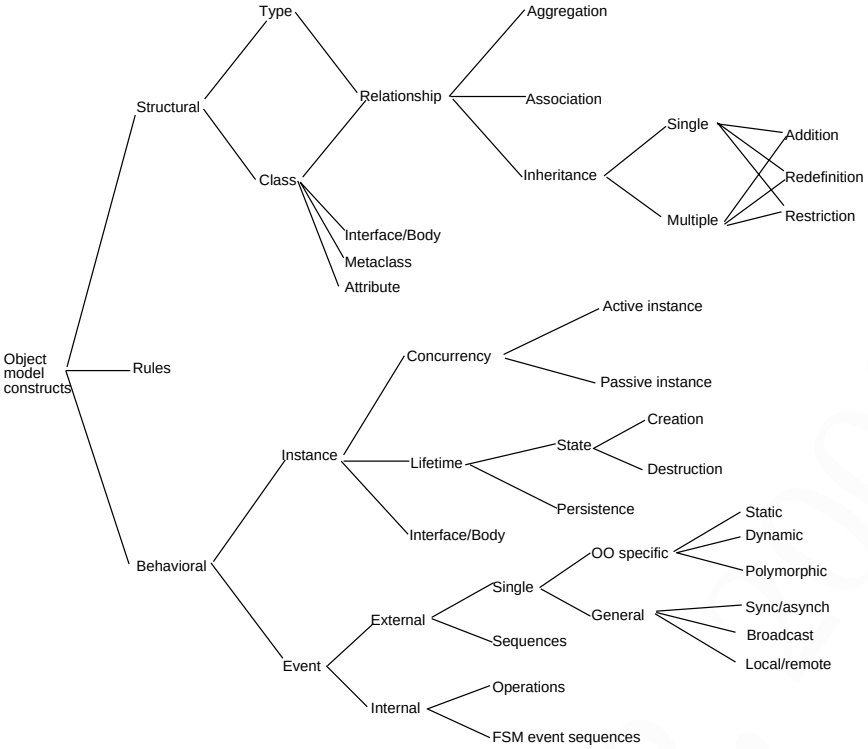


Fig. 2.22. General object model (From [396])

more than one parent, or *multiple* – where a class or type can have more than one parent. Inheritance in a class hierarchy can exhibit more features than that of a type hierarchy. Class inheritance may exhibit *addition* – where the subclass merely adds some extra properties (attributes and methods) over what is inherited from its superclass(es). Class inheritance can also involve *redefinition* – where some of the inherited properties are redefined. Class inheritance may also exhibit *restriction* – where only some properties of the superclass are inherited by the subclass. Inheritance is described in more detail in [362].

A *metaclass* is a higher-order class, responsible for describing other classes.

Rules within object-oriented modeling language are basically static rules.

Behavioral phenomena describe the dynamics of a system. Dynamic phenomena relates to *instances* of classes and the *events* or *messages* which pass between such instances. An instance has a definite *lifetime* from when it is created to when it is destroyed. In between these two events, an instance may spend time in a number of interim states. If the lifetime of an instance can exceed the lifetime of the application or process that created it, the instance is said to be *persistent*. Instances can execute in parallel (*active*) or serially

(*passive*) with others. Events are stimuli within instances. An external event is an event received by an instance. An internal event is an event generated internally within an instance which may cause a state change (through an FSM) or other action (defined by an internal operation) to be taken within the instance. Such actions may involve generating messages to be sent to other instances whereby a sequence of events (or messages) may ensue. Various mechanisms may be used to deliver a message to its destination, depending on the capabilities of the implementation language. For example, a message may employ *static* binding - where the destination is known at application compile time. Conversely, a message may employ *dynamic* binding, where the message destination cannot be resolved until application run-time. In this case, message-sending *polymorphism* may result, where the same message may be sent to more than one type (class) of instances. Messages may be categorized as either *asynchronous* where the message is sent from originator to receiver and the originator continues processing, or *synchronous* where the thread of control passes from the originating instance to the receiving instance. Messages may also be sent in *broadcast* mode where there are multiple destinations. Where an overall system is distributed among several processes, messages may be either *local* or *remote*. Many of these detailed aspects are first relevant during design of a system.

One example of the object perspective is the Object Modeling Technique(OMT).

OMT . OMT [319] have three modeling languages: the object modeling language, the dynamic modeling language, and the functional modeling language.

Object Modeling Language. This describes the static structure of the objects and their relationships. It is a semantic data modeling language. The vocabulary and grammar of the language are illustrated in Fig. 2.23.

- a) Illustrates a class, including attributes and operations. For attributes, it is possible to specify both data type and an initial value. Derived attributes can be described, and also class attributes and operations. For operations it is possible to specify an argument list and the type of the return value. It is also possible to specify rules regarding objects of a class, for instance by limiting the values of an attribute.
- b) Illustrates generalization, being non-disjoint (shaded triangle) or disjoint. Multiple inheritance can be expressed. The dots beneath *superclass2* indicates that there exist more subclasses. It is also possible to indicate a discriminator (not shown). A discriminator is an attribute whose value differentiates between subclasses.
- c) Illustrates aggregation, i.e. part-of relationship on objects.
- d) Illustrates an instance of an object and indicates the class and the value of attributes for the object.
- e) Illustrates instantiation of a class.

- f) Illustrates relationships (associations in OMT-terms) between classes. In addition to the relationship name, it is possible to indicate a role-name on each side, which uniquely identifies one end of a relationship. The figure also illustrates propagation of operations. This is the automatic application of an operation to a network of objects when the operation is applied to some starting object.
- g) Illustrates a qualified relationship. The qualifier is a special attribute that reduces the effective cardinality of a relationship. One-to-many and many-to-many relationships may be qualified. The qualifier distinguish among the set of objects at the many end of an relationship.
- h) Illustrates that also relationships can have attributes and operations. This figure also shows an example of a derived relationship (through the use of the slanted line).
- i) Illustrates cardinality constraints on relationships. Not shown in any of the figures is the possibility to define constraints between relationships, e.g. that one relationship is a subset of another.
- j) Illustrates that the elements of the many-end of a relationship are ordered.
- k) Illustrates the possibility of specifying n-ary relationships.

An example that illustrates the use of main parts of the languages is given in Fig. 2.24 indicating parts of a structural model for a conference system. A *Person* is related to one or more *Organization* through the *Affiliation* relationship. A *Person* is specialized into among others *Conference organizer*, *Referee*, *Contributer*, and *Participant*. A person can fill one or more of these roles. A *conference organizer* can be either a *OC (organizing committee)-member* or a *PC (program committee)-member* or both. A *Referee* is creating a *Review* being an *evaluation* of a *Paper*. A *PC-member* is responsible for the *Review*, but is not necessarily the *Referee*. The *Review* contains a set of *Comments*, being of a *Commenttype*. Two of the possible instances of this class "Comments to the author" and "Main contributer" is also depicted. A *Review* has a set of *Scores* being *Values* on a *Scale* measuring different *Dimensions* such as contribution, presentation, suitability to the conference and significance.

Dynamic Modeling Language. This describes the state transitions of the system being modeled. It consist of a set of concurrent state transition diagrams. The vocabulary and grammar of the language is illustrated in Fig. 2.25. The standard state transition diagram functionality is illustrated in Fig. 2.25a) and partly Fig. 2.25 b), but this figure also illustrates the possibility of capturing events that do not result in a state transition. This also includes entry and exit events for states. Fig. 2.25c) illustrates an event on event situation, whereas Fig. 2.25d) illustrates sending this event to objects of another class. Fig. 2.25e), Fig. 2.25f), and Fig. 2.25g) shows constructs similar to those found in Statecharts [161] to address the combinatorial explosion in traditional state transition diagrams. See Sect. 2.2.4 for a more detailed overview of Statecharts. Not shown in the figure are so called automatic transitions.

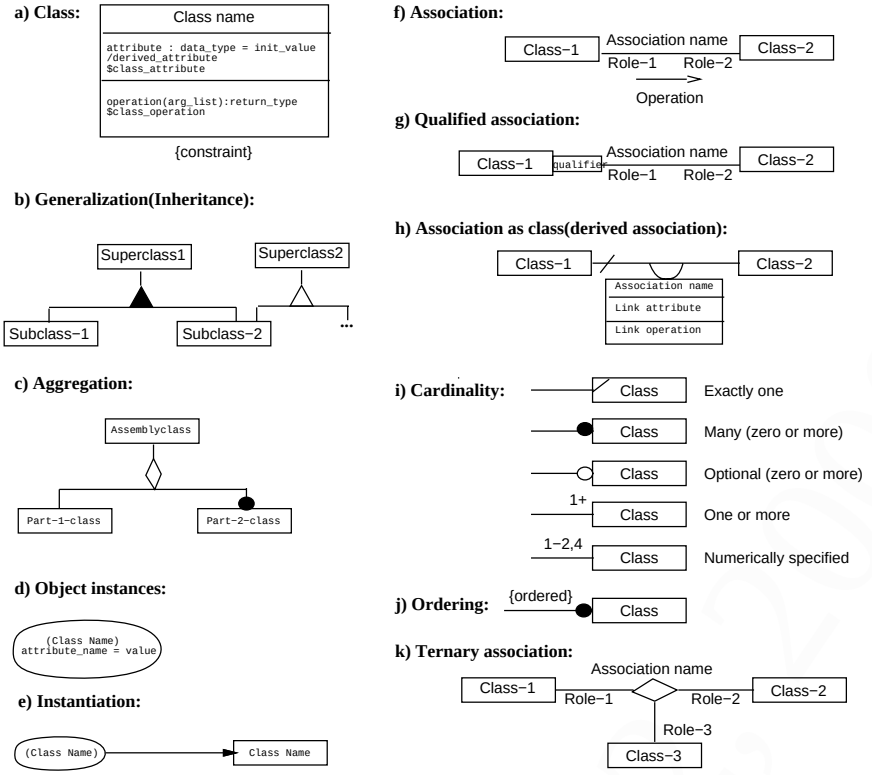


Fig. 2.23. Symbols in the OMT object modeling language

Frequently, the only purpose of a state is in this language to perform a sequential activity. When the activity is completed a transition to another state fires. This procedural way of using a state transition diagram is somewhat different from the traditional use.

Functional Modeling Language. This describes the transformations of data values within a system. It is described using data flow diagrams. The notation used is similar to traditional DFD as illustrated in Sect. 2.2.3, with the exception of the possibility of sending control flows between processes, being signals only. External agents corresponds to objects as sources or sinks of data.

A host of other object-oriented modeling languages have appeared in the literature in the late eighties and the nineties, e.g. [18, 35, 67, 68, 106, 148, 184, 197, 312, 318, 337, 401].

Overviews and comparisons of different approaches can be found in [106, 178, 396]. According to Slonim [346] "OO methodologies for analysis and design are a mess. There are over 150 contenders out there with no clear leader of the pack. Each methodology boast their own theory, their own ter-

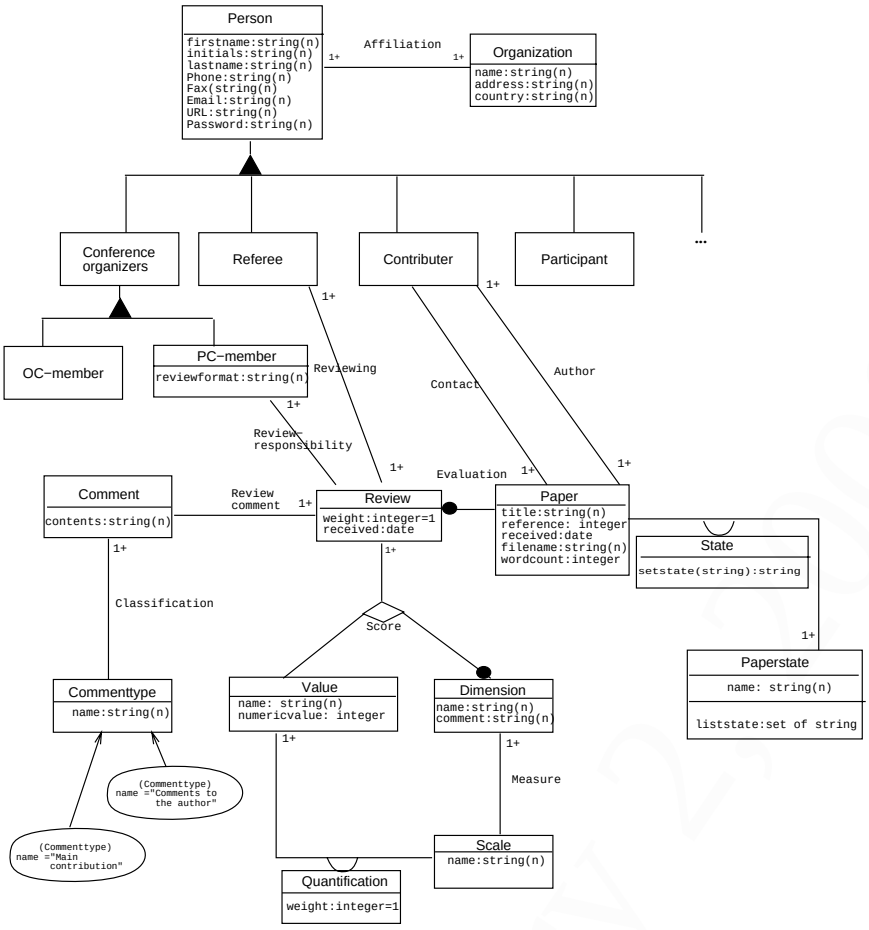


Fig. 2.24. Example of an OMT object model

minology, and their own diagramming techniques." With the recent teaming of Rumbaugh, Booch, and Jacobson on the development of UML (Unified Modeling Language) this situation might improve in the future.

We will return to other specific aspects of object-oriented modeling in Sect. 2.2.8 on the actor and role perspective.

2.2.7 The Communication Perspective

Much of the work within this perspective is based on language/action theory from philosophical linguistics. The basic assumption of language/action theory is that persons cooperate within work processes through their conversations and through mutual commitments taken within them. *Speech act theory*, which has mainly been developed by Austin and Searle [15, 327, 328]

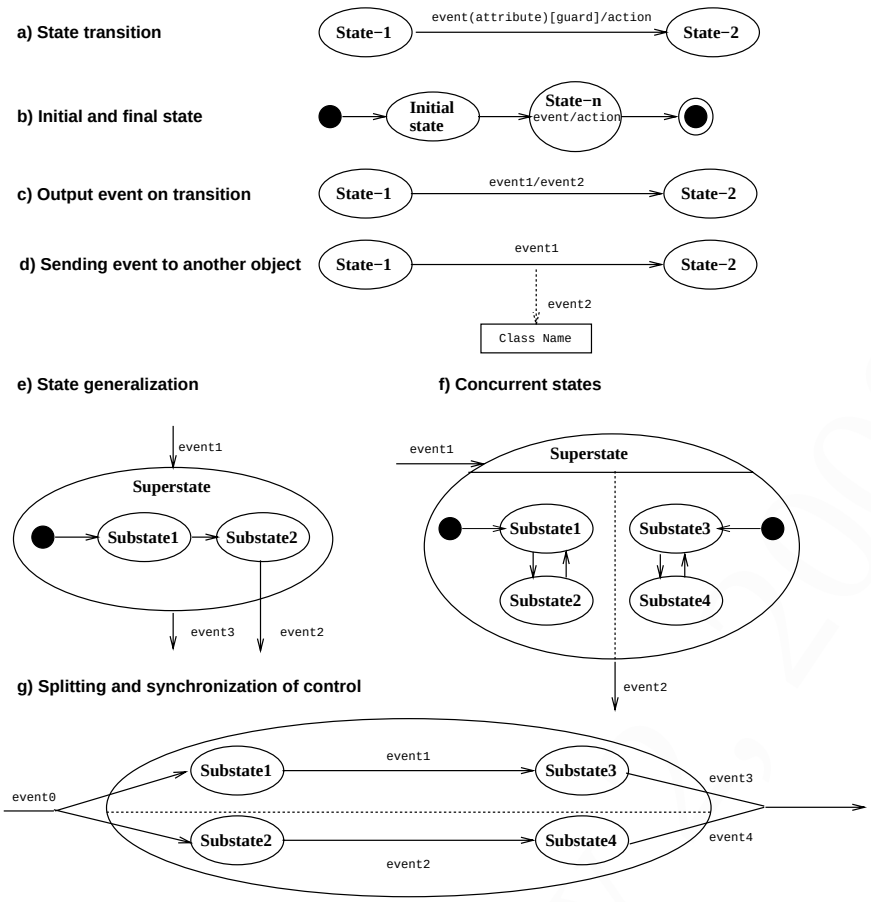


Fig. 2.25. Symbols in the OMT dynamic modeling language

starts from the assumption that the minimal unit of human communication is not a sentence or other expression, but rather the performance of certain kinds of language acts. Illocutionary logic [94, 329] is a logical formalization of the theory and can be used to formally describe the communication structure. The main parts of illocutionary logic is the illocutionary act consisting of three parts, illocutionary context, illocutionary force, and propositional context.

The context of an illocutionary act consist of five elements: Speaker (S), hearer (H), time, location, and circumstances.

The illocutionary force determines the reasons and the goal of the communication. The central element of the illocutionary force is the illocutionary point, and the other elements depend on this. Five illocutionary points are distinguished [328]:

- *Assertives*: Commit S to the truth of the expressed proposition (e.g. It is raining).
- *Directives*: Attempts by S to get H to do something (e.g. Close the window).
- *Commissives*: Commit S to some future course of action (e.g. I will be there).
- *Declaratives*: The successful performance guarantees the correspondence between the proposition p and the world (e.g. The ball is out).
- *Expressives*: Express the psychological state about a state of affairs specified in the proposition. (e.g. Congratulations!).

This distinctions is directly related to the 'direction of fit' of speech acts. We can distinguish four directions of fit.

1. Word-to-world: The propositional content of the speech act has to fit with an existing state of affairs in the world. (assertive)
2. World-to-word: The world is altered to fit the propositional content of the speech act. (directive and commissive)
3. Double direction fit: The world is altered by uttering the speech act to conform to the propositional content of the speech act. (declaratives)
4. Empty direction of fit: There is no relation between the propositional content of the speech act and the world. (expressives).

In addition to the illocutionary point, the illocutionary force contains six elements:

- Degree of strength of the illocutionary point: Indicates the strength of the direction of fit.
- Mode of achievement: Indicates that some conditions must hold for the illocutionary act to be performed in that way.
- Propositional content conditions: E.g. if a speaker makes a promise, the propositional content must be that the speaker will cause some condition to be true in the future.
- Preparatory condition: There are basically two types of preparatory conditions, those dependant on the illocutionary point and those dependant on the propositional content.
- Sincerity conditions: Every illocutionary act expresses a certain psychological state. If the propositional content of the speech act conforms with the psychological state of the speaker, we say that the illocutionary force is sincere.
- Degree of strength of sincerity condition: Often related to the degree of strength of the illocutionary point.

Speech acts are elements within larger conversational structures which define the possible courses of action within a conversation between two actors. One class of conversational structures are what Winograd and Flores [400] calls 'conversation for action'. Graphs similar to state transition diagrams have been used to plot the basic course of such a conversation (see Fig. 2.26). The

conversation start with that part A comes with a request (a directive) going from state 1 to state 2. Part B might then promise to fulfill this request performing a commissive act, sending the conversation to state 3. Alternatively , B might the decline the request, sending the conversation to the end-state 8, or counter the request with an alternative request, sending the conversation into state 6. In a normal conversation, when in state 3, B reports completion, performing an assertive act, the conversation is sent to state 4. If A accept this, performing the appropriate declarative act, the conversation is ended in state 5. Alternatively, the conversation is returned to state 3.

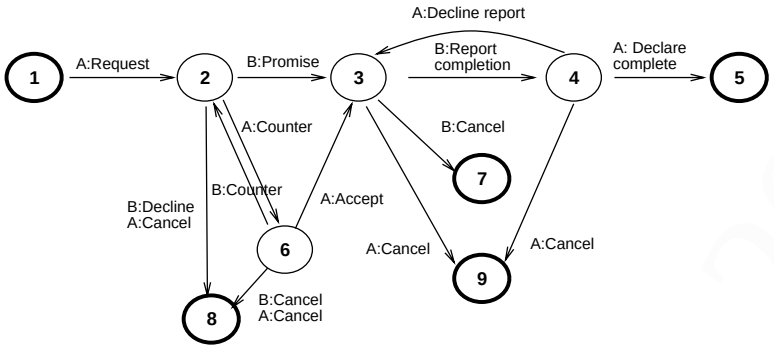


Fig. 2.26. Conversation for action (From [400])

This is only one form of conversation. Several others are distinguished, including conversations for clarification, possibilities, and orientation.

This application of speech act-theory forms the basis for several computer systems, the best known being the Coordinator [118].

Speech act theory is often labeled as a 'meaning in use theory' together with the philosophy of the later Wittgenstein. Both associate the meaning of an expression with how it is used. However, it is also important to see the differences between the two. Searle associated meaning with a limited set of rules for how an expression should be used to perform certain actions. With this as a basis, he created a taxonomy of different types of speech acts. For Wittgenstein, on the other hand, meaning is related to the whole context of use and not only a limited set of rules. It can never be fully described in a theory or by means of systematic philosophy

Speech act theory is also the basis for modeling of work-flow as coordination among people in Action Workflow [260]. The basic structure is shown in Fig. 2.27.

Two major roles, customer and supplier, are modeled. Work-flow is defined as coordination between actors having these roles, and is represented by a conversation pattern with four phases. In the first phase the customer makes a request for work, or the supplier makes an offer to the customer.

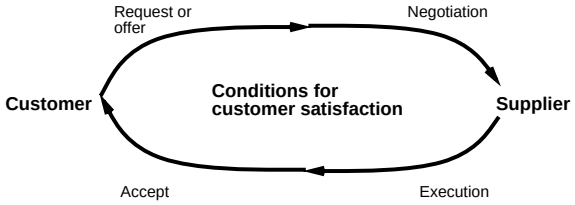


Fig. 2.27. Main phases of action workflow

In the second phase, the customer and supplier aims at reaching a mutual agreement about what is to be accomplished. This is reflected in the contract conditions of satisfaction. In the third phase, after the performer has performed what has been agreed upon and completed the work, completion is declared for the customer. In the fourth and final phase the customer assess the work according to the conditions of satisfaction and declares satisfaction or dissatisfaction. The ultimate goal of the loop is customer satisfaction. This implies that the work-flow loop have to be closed. It is possible to decompose steps into other loops. The specific activities carried out in order to meet the contract are not modeled. The four phases in Fig. 2.27 corresponds to the "normal path" 1-5 in Fig. 2.26.

Some newer approaches to workflow modeling include aspects of both the functional (see Sect. 2.2.3) and language action modeling. In WooRKS [2] functional modeling is used for processes and LA for exceptions thus not using these perspectives in combination. TeamWare Flow [361] and Obligations [33] on the other hand can be said to be hybrid approaches, but using radically different ontologies from those found in traditional conceptual modeling.

Habermas took Searle's theory as a starting point for his theory of communicative action [154]. Central to Habermas is the distinction between strategic and communicative action. When involved in strategic action, the participants strive after their own private goals. When they cooperate, they are only motivated empirically to do so: they try to maximize their own profit or minimize their own losses. When involved in communicative action, the participants are oriented towards mutual agreement. The motivation for cooperation is thus rational. In any speech act the speaker S raises three claims: a claim to truth, a claim to justice, and a claim to sincerity. The claim to truth refers to the object world, the claim to justice refers to the social world of the participants, and the claim to sincerity refers to the subjective world of the speaker. This leads to a different classification of speech acts [92]:

- Imperativa: S aims at a change of the state in the objective world and attempts to let H act in such a way that this change is brought about. The dominant claim is the power claim. Example; " I want you to stop smoking"
- Constativa: S asserts something about the state of affairs in the objective world. The dominate claim is the claim to truth. Example: "It is raining"

– Regulative: S refers to a common social world, in such a way that he tries to establish an interpersonal relation which is considered to be legitimate. The dominant claim is the claim to justice. Example: "Close the window", "I promise to do it tomorrow".
– Expressiva: S refers to his subjective world in such a way that he discloses publicly a lived experience: The dominant claim is the claim to sincerity. Example: "Congratulations" .

A comparisons between Habermas' and Searle's classifications is given in Fig. 2.28.

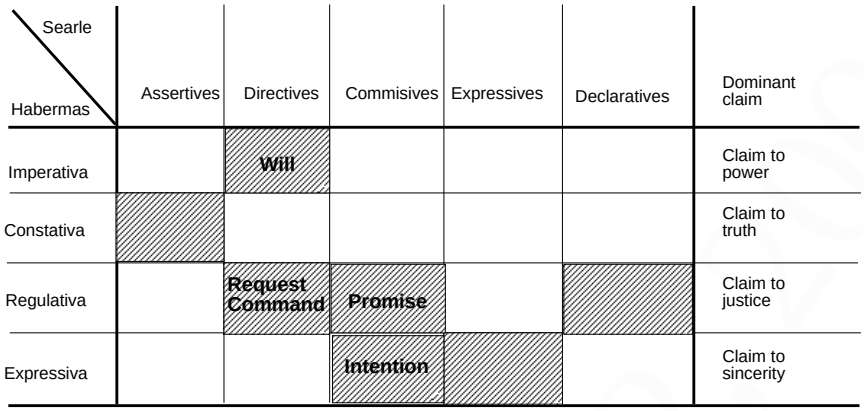


Fig. 2.28. Comparing communicative action in Habermas and Searle (From [92])

In addition to the approach to workflow-modeling described above, several other approaches to conceptual modeling are inspired by the theories of Habermas and Searle such as COMMODIOUS [172], SAMPO [14], and ABC/DEMO. We will describe one of these here, ABC.

ABC-diagrams. Dietz [91] differentiate between two kinds of conversations:

– Actagenic, where the result of the conversation is the creation of something to be done (agendum), consisting of a directive and a commissive speech act.
– Factagenic, which are conversations which are aimed at the creation of facts typically consisting of an assertive and a declarative act.

Actagenic and factagenic conversations are both called performative conversations. Opposed to these are informative conversations where the outcome is a production of already created data. This includes the deduction of data using e.g. derivation rules.

A transaction is a sequence of three steps (see Fig. 2.29): Carrying out an actagenic conversation, executing an essential action, and carrying out a factagenic conversation. In the actagenic conversation initiated by subject A,

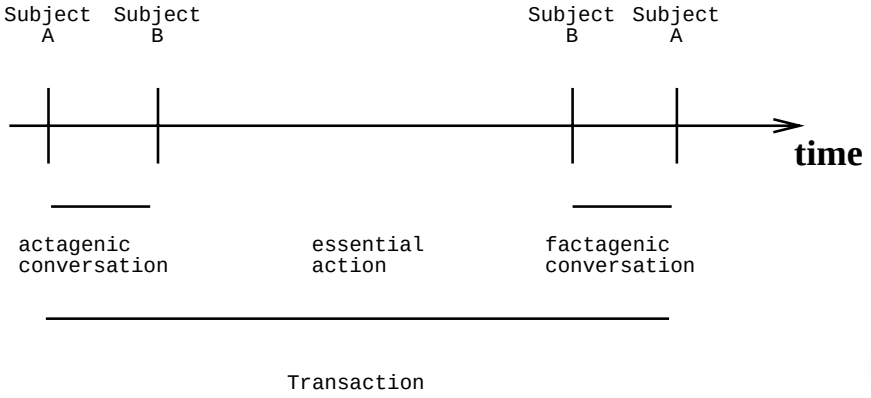


Fig. 2.29. The pattern of transaction

the plan or agreement for the execution of the essential action by subject B is achieved. The actagenic conversation is successful if B commits himself to execute the essential action. The result then is an agendum for B.

An agendum is a pair $< a, p >$ where a is the action to be executed and p the period in which this execution has to take place.

In the factagenic conversation, the result of the execution are stated by the supplier. It is successful if the customer accepts these results. Note the similarities between this and the workflow-loop in action workflow.

In order to concentrate on the functions performed by the subjects while abstracting from the particular subjects that perform a function, the notion of actor is introduced. An actor is defined by the set of actions and communications it is able to perform.

The actor that initiates the actagenic conversation and consequently terminate the factagenic one of transactions of type T, is called the initiator of transaction type T. Subject B in Fig. 2.29 is called the executor of transaction T.

An actor that is element of the composition of the subject system is called an internal actor, whereas an actor that belongs to the environment is called an external actor. Transaction types of which the initiator as well as the executor is an internal actor is called an internal transaction. If both are external, the transaction is called external. If only one of the actors is external it is called an interface transaction type. Interaction between two actors takes place if one of them is the initiator and the other one is the executor of the same transaction type. *Interstriction* takes place when already created data or status-values of current transactions are taken into account in carrying out a transaction.

In order to represent interaction and interstriction between the actors of a system, Dietz introduce ABC-diagrams. The graphical elements in this language are shown in Fig. 2.30. An actor is represented by a box, identified by

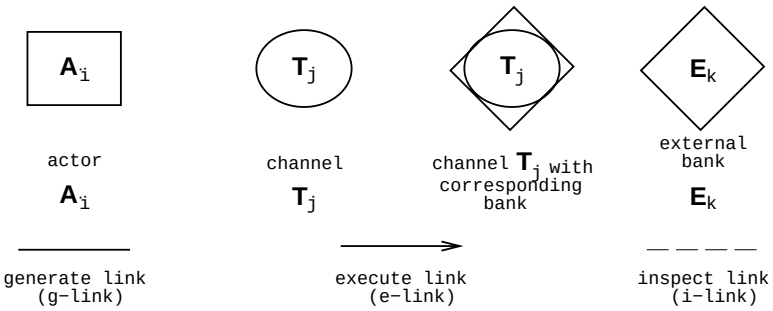


Fig. 2.30. The symbols of the ABC-language (From [91])

a number. A transaction type is represented by a disk. The operational interpretation of a disk is a store for the statuses through which the transaction of that type pass in the course of time. The disk symbol is called a channel. The diamond symbol is called a bank, and contain the data created through the transaction. The actor who is the initiator of a transaction type is connected to the transaction channel by a generate link (g-link) symbolized by a plain link. The actor who is the executor is connected to the transaction by an execute link (e-link). Informative conversations are represented by inspect links (i-links), symbolized by dashed lines.

In [376], it is in addition illustrated how to show the sequence of transactions in a transaction sequence graph. It is also developed a transaction process model which is an extension of the model presented in Fig. 2.26 including an indication of the dominant claim of the conversation that is potentially countered.

2.2.8 The Actor and Role Perspective

The main phenomena of languages within these perspective are actor (alternatively agent) and role. The background for modeling of the kind described here comes both from work on (object-oriented) programming languages (e.g actor-languages [371]), and work on intelligent agents in artificial intelligence (e.g [133, 339]).

ALBERT (Agent-oriented Language for Building and Eliciting Real-Time requirements) [98, 99] have a set of specification language for modeling complex real-time cooperative distributed systems which are based on describing a system as a society of agents, each of them with their own responsibilities with respect to the actions happening in the system and its time-varying perception of the behavior of the other agents. A variety of requirements can be described with ALBERT, such as structural, temporal, functional, behavioral, in addition to real-time and cooperative aspects which are covered through the modeling of distributed systems in terms of agents, each

of them characterized with time-varying communication possibilities. Communication mechanisms allow to describe how an agent perceive data made available to it by other agents and show parts of its data to other agents. We will here concentrate on the agent modeling aspect of ALBERT.

Agents, as defined in ALBERT, may be seen as a specialization of objects. Models are made at two levels.

- Agent level: A set of possible behaviors are associated with each agent without any regard to the behavior of other agents
- Society level: Interactions between agents are taken into account and lead to additional restrictions on the behavior of each individual agent.

The formal language is based on a variant of temporal logic extended with actions, agents, and typical patterns of constraints. The declaration of agents consist in the description of the state structure and the list of the actions its history can be made of. The state is defined by its components which can be individuals collections of individuals. Components can be time-varying or constant. Agents include a key mechanism that allows the identification of the different instances. A type is automatically associated to each class of agents. Figure 2.31 shows the model associated with the declaration of the state structure of a cell (a part of a CIM production system).

Sets and instances are depicted as small rectangles with rectangles inside indicating the type (e.g. Out-full of type BOOLEAN, or Input-stock of type RIVET). Actions are depicted as small rectangles with ovals inside (e.g. Remove-bolt). Actions might have arguments (e.g. BOLT of Remove-bolt). A wavy line between components expresses that the value of a component may be derived from others (e.g. Output-stock from Out-full). It is possible to distinguish between internal and external action and to express the visibility relationships linking the agent to the environment. The components within the parallelogram is under the control of the described agent while information outside denotes elements which are imported from other agents of the society the agent belongs to. Boxes within the parallelogram with an arrow going out from them denote that data is exported to the outside (e.g. Output-stock to Manager).

Agents are grouped into societies, which themselves can be grouped into other societies. The existing hierarchy of agents are expressed in term of two combinations: Cartesian product and set. Constraints are used for pruning the infinite set of possible lives of an agent. These are divided into ten headings and three families to provide methodological guidance. The families are:

- Basic constraints: Used to describe the initial state of an agent, and to give the derivation rules for derived components.
- Local constraints: Related to the internal behavior of the agent.
- Cooperative constraints: Specifies how the agent interacts with its environment.

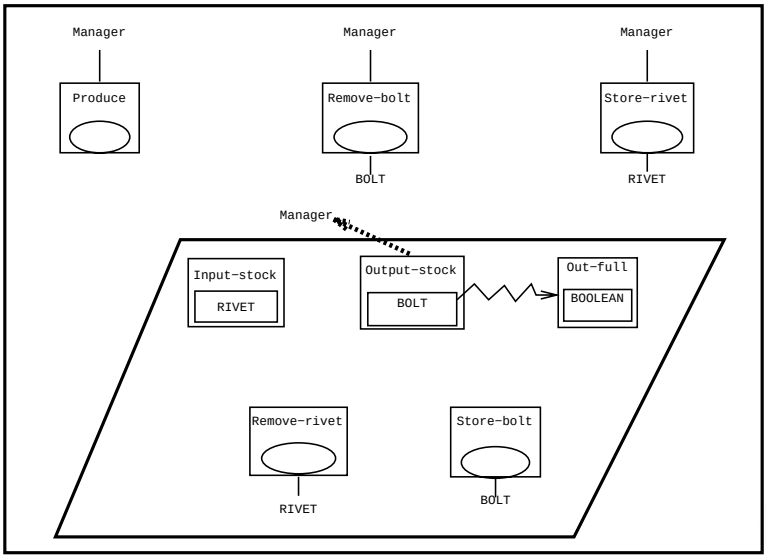


Fig. 2.31. Example of an ALBERT model (From [99])

Organizational modeling: Yu and Mylopoulos [406, 407] have proposed a set of integrated languages to be used for organizational modeling:

– The Actor Dependency modeling language.
– The Agents-Roles-Positions modeling language.
– The Issue-Argumentation modeling language.

The Issue-Argumentation modeling language is an application of a subset of the non-functional framework presented in Sect. 2.2.5. The two other modeling languages are presented below.

In actor dependency models each node represent a social actor/role. Figure 2.32 gives an example of such a model depicting the goods acquisition of a company. The actors/roles here are *purchasing*, *client*, *receiving*, *vendor*, and *accounts payable*. Each link between the nodes indicates that a social actor depends on the other to achieve a goal. The depending actor is called the *depender*, and the actor that is depended upon is called the *dependee*. The object assigned to each link is called a *dependum*. It is distinguished between four types of dependencies:

– Goal dependency: The depender depends on the dependee to bring about a certain situation. The dependee is expected to make whatever decisions are necessary to achieve the goal. In the example, the client just wants to have the item, but does not care how the purchasing specialist obtains price quotes, or which supplier he orders from. Purchasing, in turn, just wants the vendor to have the item delivered, but does not care what mode of transportation is used etc.

- Task dependency: The depender depends on the dependee to carry out an activity. A task dependency specifies how, and not why the task is performed. In the example, purchasing's dependency on receiving is a task dependancy because purchasing relies on receiving to follow procedures such as: Accept only if the item was ordered. Similarly, the client wants accounts payable to pay only if the item was ordered and has been received.
- Resource dependency: The depender depends on the dependee for the availability of some resources (material or data). Accounting's dependencies for information from purchasing, receiving, and the vendor before it can issue payment are examples of resource dependencies.
- Soft-goal dependencies: Similar to a goal dependency, except that the condition to be attained is not accurately defined. For example, if the client wants the item promptly, the meaning of promptly needs to be further specified.

The language allows dependencies of different strength: Open, Committed, and Critical. An activity description, with attributes as input and output, sub-activities and pre and post-conditions expresses the rules of the situation. In addition to this, goal attributes are added to activities. Several activities might match a goal, thus subgoals are allowed.

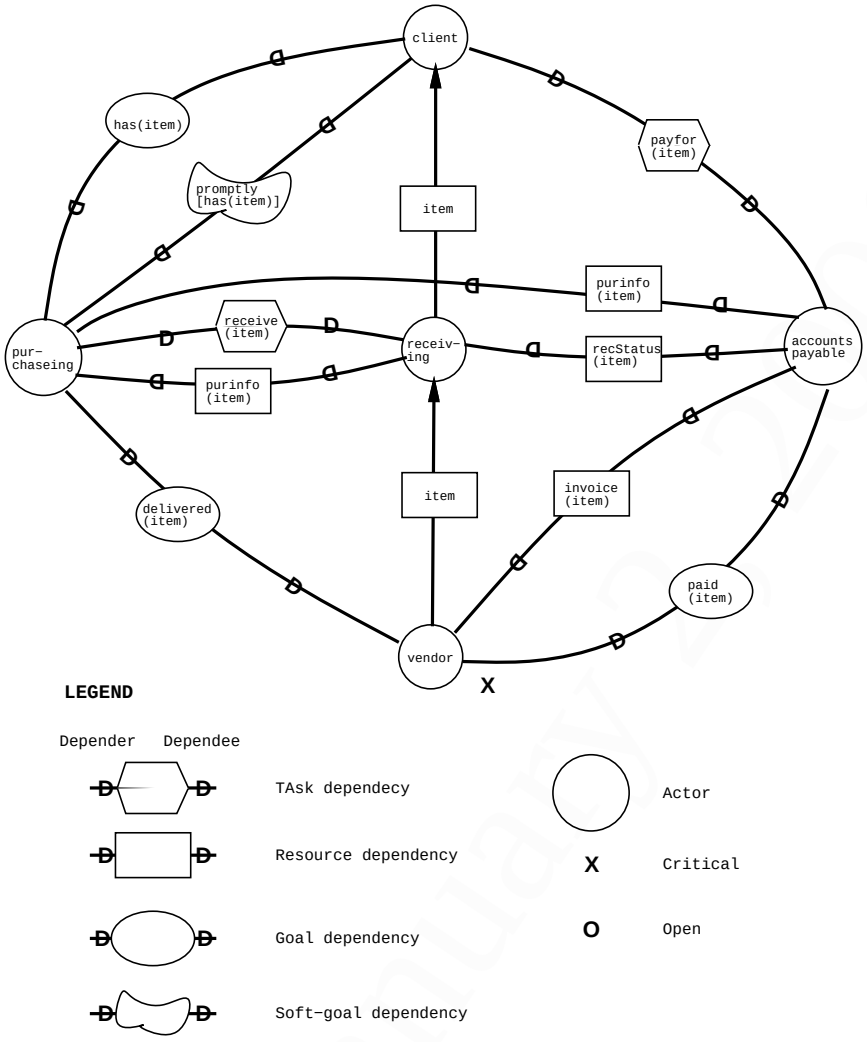


Fig. 2.32. Example of an actor dependency model (From [407])

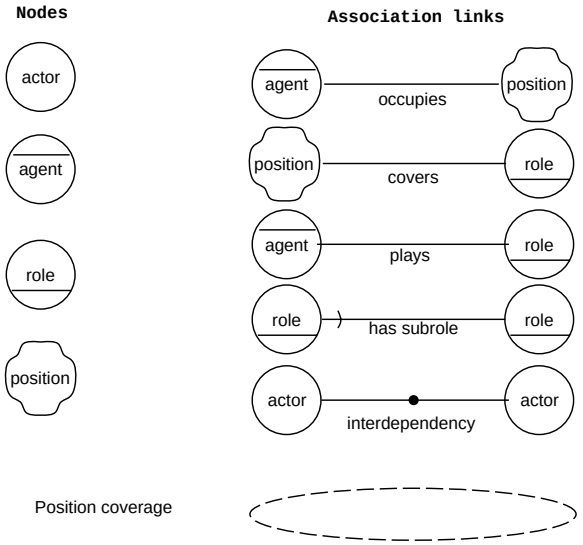


Fig. 2.33. Symbols in agents-role-position modeling language (From [406])

The Agents-Roles-Positions modeling language consists of a set nodes and links as illustrated in Fig. 2.33. An *actor* is here as above used to refer to any unit to which intentional dependencies can be ascribed. The term *social actor* is used to emphasize that the actor is made up of a complex network of associated agents, roles, and positions. A *role* is an abstract characterization of the behavior of a social actor within some specialized context or domain. A *position* is an abstract place-holder that mediates between agents and roles. It is a collection of roles that are to be played by the same agent. An *agent* refers to those aspects of a social actor that are closely tied to its being a concrete, physically embodied individual.

Agents, roles, and positions are associated to each other via links: An agent (e.g. John Krogstie) can *occupy* a position (e.g. program coordinator) , a position is said to *cover* a role (e.g. program coordinator covers delegation of papers to reviewers), and an agent is said to *play* a role. In general these associations may be many-to-many. An *interdependency* is a less detailed way of indicating the dependency between two actors. Each of the three kinds of actors- agents, roles, and positions, can have sub-parts.

OORASS - Object oriented role analysis, synthesis and structuring. OORASS [312] is really a pure object-oriented method, but we have chosen to present it here since what is special to OORASS is the modeling of roles.

A role model is a model of object interaction described by means of message passing between roles. It focuses on describing patterns of interaction without connecting the interaction to particular objects.

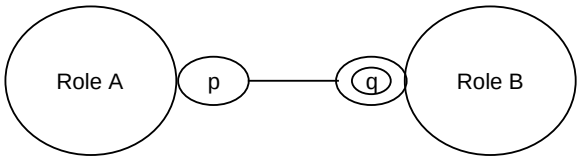


Fig. 2.34. Symbols in the OORASS role interaction language

The main parts of a role model is described in Fig. 2.34. A *role* is defined as the why-abstraction. Why is an object included in the structure of collaborating objects? What is its position in the organization, what are the responsibilities and duties? All objects having the same position in the structure of objects play the same role. A role only has meaning as a part of some structure. This makes the role different from objects which are entities existing "in their own right". An object has identity and is thus unique, a role may be played by any number of objects (of any type). An object is also able to play many different roles. In the figure there are two roles A and B. A path between two roles means that a role may 'know about' the other role so that it can send messages to it. A path is terminated by a port symbol at both ends. A port symbol may be a single small circle, a double circle, or nothing. Nothing means that the near role do not know about the far role. A single circle (p) indicates that an instance of the near role (A) knows about none or one instance of the far role (B). A double circle (q) indicates that an instance of the near role knows about none, one or more instances of the far role. In the figure 'p' is a reference to some object playing the role B. Which object this is may change during the lifetime of A. If some object is present, we are always assured that it is capable of playing the role B. For a port, one can define an associated set of operations called a contract. These operations are the ones that the near role requires from the far role, not what the near role implements. The signatures offered must be deduced from what is required in the other end.

Role models may be viewed through different *views*.

- Environment view: The observer can observe the system interact with its environment.
- External view: The observer can observe the messages flowing between the roles.
- Internal view: The observer can observe the implementation

Other views are given in OORASS using additional languages with structural, functional, and behavioral perspectives.

2.3 Applying Several Modeling Perspectives

We have above presented different perspectives towards conceptual modeling. Based on social construction theory, the general features of the world can not be said to exist a priori. According to this belief one might wish to go to the other extreme — an approach without any presumptions at all. However, this is impossible. Any methodology and any language implies some presumptions. Thus, having an approach totally free of presumptions would mean to have no approach at all, inventing a new one fit for the specific problem for every new development and maintenance task. For philosophers this might be acceptable, but engineers are expected to adapt to certain demands for efficiency. Inventing a new approach for every development and maintenance effort would not give us that efficiency, neither is it likely that it will give better CIS-support for the organization. Developing and maintaining a CIS without any fixed ideas about how it should be done would be tedious and unsystematic – as stated by Boehm [32], the ad hoc methods used in the earliest days of software development were much worse than those used today. So clearly one needs to make some presumptions, one need to have some fixed ideas. What is necessary is to find a *point of balance* — making enough presumptions for the approach to be systematic and efficient, but not so many that its flexibility and applicability is severely reduced. We can become aware of some of our presumptions, and in that way emancipate ourselves from some of the limits they place on our thinking, but we can never free us from all presumptions.

As we have illustrated in this chapter, there are a number of different approaches to conceptual modeling, each emphasizing different aspects of the perceived reality. Several researchers have claimed that one perspective is better, or more natural, than others:

- Sowa [352] bases his language for conceptual graphs on work on human perception and thinking done in cognitive psychology, and uses this to motivate the use of the language. It seems safe to say that even with his convincing discussion, conceptual graphs have had a very limited influence on conceptual modeling practices and the development and maintenance of CISs in most organizations, even if its has received much attention within computer science research[5].
- In the last years, many authors have advocated object-oriented modeling partly based on the claim that it is a more natural way to perceive the world [244, 396]. The view that object-orientation is a suitable perspective for all situations have been criticized by many in the last couple of years; see e.g. [43, 173, 183]. The report on the First International Symposium on Requirements Engineering [183] said it so strongly that "requirements are not object-oriented. Panelist reported that users do not find it natural to express their requirements in object-oriented fashion". Even if there

[5] The third international conference on the topic was held in August 1995.

are cases where a purely object-oriented perspective is beneficial, it does not seem to be an appropriate way of describing all sorts of problems, as discussed in [173]. Newer approaches to OOA claim to attack some of these problem, see e.g. [106]. In any case, as stated by Meyer [262], "Object technology is not about modeling the real world. Object technology is about producing quality software, and the way to obtain this is to devise the right abstractions, whether or not they model what someone sees as the reality".
- In Tempora [366], rules were originally given a similar role in that it was claimed that "end users perceive large parts of a business in terms of policies or rules". This is a truth with modification. Even if people may act according to rules, they are not necessarily looking upon it as they are as discussed by Stamper [354]. Rule-based approaches also have to deal with several deficiencies, as discussed earlier in the chapter.
- Much of the existing work on conceptual modeling that has been based on a constructivistic world-view has suggested language/action modeling as a possible cornerstone of conceptual modeling [142, 203, 400], claiming that it is more suitable than traditional "objectivistic" conceptual modeling. On the other hand, the use of this perspective has also been criticized, also from people sharing a basic constructivistic outlook. An overview of the critique is given in [83]:
  - Speech act theory is wrong in that it assumes a one-to-one mapping between utterances and illocutionary acts, which is not recognizable in real life conversations.
  - The normative use of the illocutionary force of utterances is the basis for developing tools for the discipline and control over organizations member's actions and not supporting cooperative work among equals.
  - The language/action perspective does not recognize that embedded in any conversation is a process of negotiating the agreement of meaning.
  - The language/action perspective misses the locality and situatedness of conversations, because it proposes a set of fixed models of conversations for any group without supporting its ability to design its own conversation models.
  - The language/action perspective offers only a partial insight; it has to be integrated with other theories.
- As discussed earlier in this chapter, also functionally and structurally oriented approaches have been criticized in the literature [46, 287].

Although the use of a single perspective has been criticized, this does not mean that modeling according to a perspective should be abandoned, as long as we do not limit ourselves to one single perspective. A model expressed in a given language emphasize a specific way of ordering and abstracting ones internal reality. One model in a given language will thus seldom be sufficient. With this in mind more and more approaches are based on the combination of several modeling languages. There are at least four general ways of attacking this:

1. Use existing single-perspective languages as they are defined, without trying to integrate them further. This is the approach followed in many existing CASE-tools.
2. Refine common approaches to make a set of formally integrated, but still partly independent set of languages.
3. Develop a set of entirely new integrated conceptual modeling languages.
4. Create frameworks that can be used for creating the modeling languages that are deemed necessary in any given situation.

A consequence of a combined approach is that it requires much better tool support to be practical. Due to the increased possibilities of consistency checking and traceability across models, in addition to better possibilities for the conceptual models to serve as input for code-generation, and to support validation techniques such as execution, explanation generation, and animation the second of these approaches has been receiving increased interest, especially in the academic world. Basing integrated modeling languages on well-known modeling languages also have advantages with respect to perceptibility, and because of the existing practical experience with these languages. Also many examples of the third solution exist, e.g. ARIES [190] and DAIDA [187], and of the fourth e.g. [279, 288] together with work on so-called meta-CASE systems e.g. [249, 378]. Work based on language-modeling might also be used to improve the applicability of approaches of all the other types.

In the next section, we will present a comparison of the expressiveness of a set of conceptual modeling languages. Then, in the last section of this chapter we will present an approach to modeling that can be used according to all the above mentioned perspectives. The approach is called PPP and is developed at the information system group at IDI, NTNU. We will throughout the book return to this approach for exemplifying different techniques for conceptual modeling. In this chapter, we only present the language aspects.

2.4 On the Expressiveness of CMLs

What constitutes a good modeling language will be discussed in more detail in Sect. 3.11. In this section, we will concentrate on the expressiveness of CMLs, and review some analysis on this subject. We have in generally retained the terminology of the different approaches, thus terms are partly used differently here than as defined in Appendix D.

In the mid 1980's, there was considerable interest in analyzing and comparing different modeling languages and methodologies, as exemplified by the IFIP conferences [285] and [283]. The analyses have all been aimed at increasing the understanding of conceptual modeling, and of the expressiveness needed by CMLs. In addition, the works we refer to here other specific motivations:

- Wand and Weber have developed an *ontological model* of information systems (e.g. [387, 388]), which they exploit for evaluation of the expressiveness of CMLs.
- IFIP working group 8.1 has developed a methodology framework from which components may be selected to define new languages and methodologies [284].
- Several projects have been aiming at developing multi-language environments to let developers choose languages according to the problem or to personal preferences. Common to such systems is the use of a highly expressive internal language serving as a bridge in the translations between different external languages. Examples presented here, are AMADEUS [28, 72], GDR [245], and ARIES [190]. Note that such systems are developed from the recognition that different languages are often very similar in the meaning of the constructs offered.
- Hull and King present a unified data model for a survey of semantic data models in [175].

The last three involve development of what we may call unified languages. The approach we will take in the following is to represent the essential parts of the works listed above in meta-models, using the language depicted in Fig. 2.35. Although we only focus on the major constructs, the meta-models will serve as a basis for comparison of the different results, and give us useful knowledge about the needed constructs of CMLs.

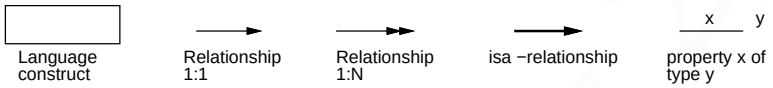


Fig. 2.35. A data modeling language used for meta-modeling

2.4.1 The Ontological Model of Information Systems

The ontological model has its origin in general systems theory. As an information system indeed is a system, it is assumed that systems theory can be used for analysis and design of information systems in particular. The term 'ontological' indicates that the model is concerned only with essential aspects of systems, those which convey their *deep structures*. An information system is considered to be a model of a real world system, and its goodness is measured by the extent it represents the meaning of the real world system. Deep structures are seen as opposed to *surface structures*, which describe the system appearance for and interaction with its users, and the *physical structures* which deal with technological aspects and implementation. The major constructs of the ontological model are depicted in Fig. 2.36.

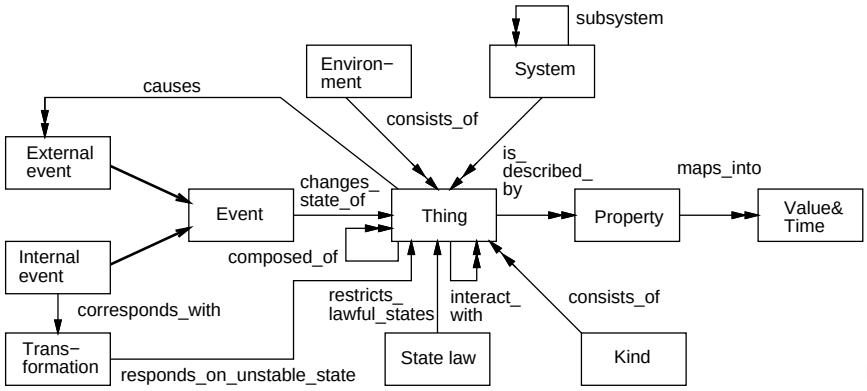


Fig. 2.36. A metamodel of Wand and Weber's ontology

The most central phenomena in the ontological model are *thing, property, state*, and *transformation*. From these, all other constructs can be derived. Things are what the world is made up of. Things may be composite, consisting of other things. Things are described by properties, that map them into values. A *kind* is a set of things with two or more common properties. The state of a thing at a particular point in time is the vector of values of its properties. A *state law* restricts the states of a thing to a set of states which are deemed lawful in some sense. A *system* is a set of things which *interact*, i.e. their states affect the states of other things in the system. A system can be decomposed into *subsystems*. The *environment* consists of things which interact with the things in the system, in the way that they may directly change the state of a thing through an *external event*. Such an event may lead the system to an *unstable state*, to which *transformations* respond by bringing the system back to a stable state.

The ontological model applies at all phases of system development. Models from different phases should preserve *invariants* for the final implemented system to be a good representation of the initial real world system. The ontological model can be used to assess the *ontological completeness* of different CMLs, to compare different CMLs, and its foundation in systems theory has been exploited to analyze decompositions of systems.

2.4.2 A Methodology Framework

Olle et al. present a comprehensive methodology framework in [284]. This framework is the result of joint work of participants in the IFIP working group 8.1, and builds on the authors' knowledge of a large number of existing methodologies. As with other frameworks, this one also divides development into phases, of which *business analysis*, *system design* and *construction design* are considered in depth, and focuses on the delivered components from

each phase. These should cover the three perspectives of data, process and behavior, as well as the integration of these perspectives.

The detailed descriptions of components give normative guidelines for what models of information systems should represent. In the following, we will focus on business analysis and the components delivered from this phase. In Fig. 2.37, a simplified metamodel corresponding to the framework is depicted. The simplifications made should not exclude any essential components.

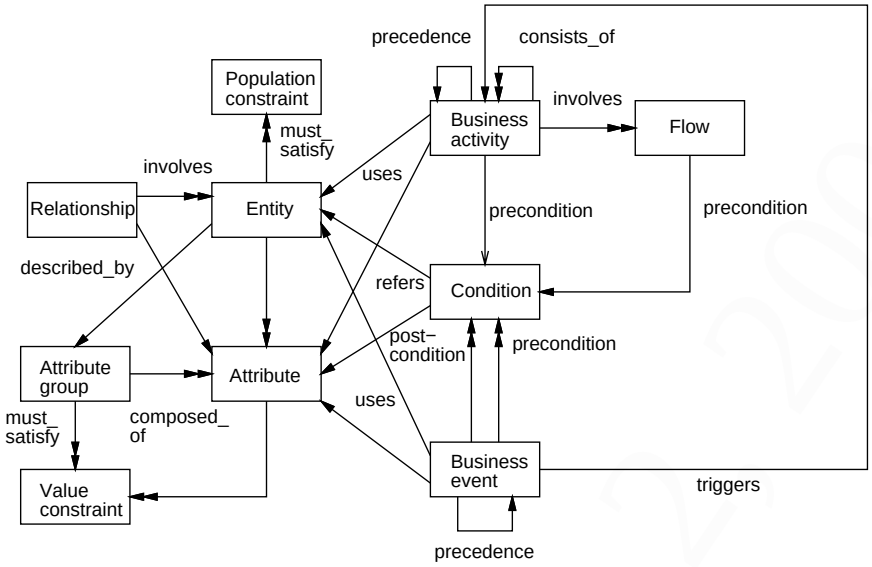


Fig. 2.37. A metamodel corresponding to the methodology framework

Static aspects of a business are described by *entities*, *relationships*, *attributes* and *constraints*. Entities are described by name, whereas relationships are described by name, class (unary, binary, n-ary) and type (cardinalities). Attributes define the state space for entities, but can also describe relationships. Attributes may be organized into *groups*. Constraints are either *value constraints* or *population constraints*. Value constraints can be uniqueness constraints, referential constraints or general check constraints. Population constraints involve overlap of populations of different entity types.

In the process perspective, the focus is on *business activities*. These can be decomposed to a number of sub-activities, and precedence relationships exist among activities at the same decomposition level. Activities receive *flows* of information or material, and produce flows as well. They can be started if certain *preconditions* hold.

The behavior perspective is covered through *business events*, which are events being perceived as pertinent to the business. They have an event name, and there may exist certain precedence relationships among them. Also, a

precondition may need to be satisfied for an event to take place, and a post-condition may need to be satisfied after an event has occurred.

The perspectives are integrated in the following manner: Activities and events use entities and attributes, in the sense that they may refer and change their states. Conditions refer to entities and attributes. Business events may trigger activities.

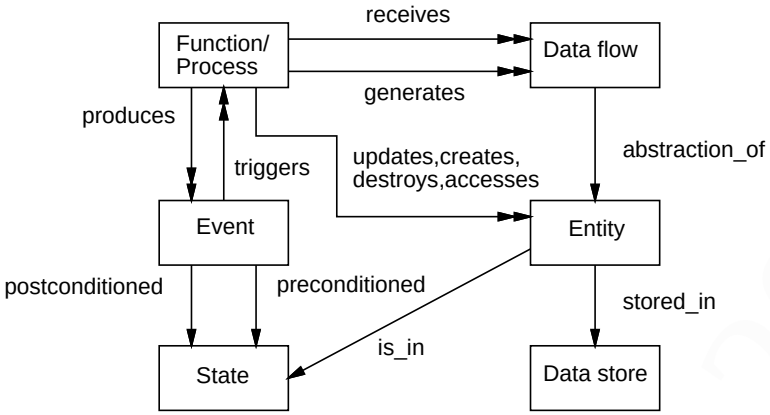


Fig. 2.38. The unified model in AMADEUS

2.4.3 The AMADEUS Metamodel

In the ESPRIT-project AMADEUS, an attempt was made to develop a unified metamodel for language integration. The approach taken was to analyze a set of ten well-known and representative modeling languages, identify the needed basic constructs in a unified metamodel, and then provide a general representation of this model. The surveyed languages included JSD [51], NIAM [273], SSADM [95], IE [185], SADT, and ISAC [246]. In [72], the work is presented in some detail. Here, we only present the main result, i.e. the unified metamodel. It has the constructs shown in Fig. 2.38.

As can be seen from the model, six main constructs are identified from the analysis of the languages; *function/process, data flow, entity, event, state* and *store*. Their relationships are also represented in the figure. For instance, a process is triggered by an event, it receives and generates data flows, manipulates entities and produces new events.

A frame based representation language (UMRL) is employed to represent the unified metamodel. The frame language has the standard frame-slot-facet constructs, and the mappings from the unified metamodel to UMRL are as described in the following. First, a construct in the unified metamodel is represented as a frame. No other frames are allowed. Second, a relationship in

the unified model is represented as a slot in UMRL. In addition to these slots, slots to define part-subpart relationships, instance relationships and generalization relationships are allowed. Finally, any value-facet of UMRL refers to another frame. Other facets describe properties of slots like cardinalities, range, conditions etc.

Using this internal representation, principles for mapping rules between models are given, via mappings to the unified metamodel. Hence, an integration of CMLs in CASE environments is facilitated.

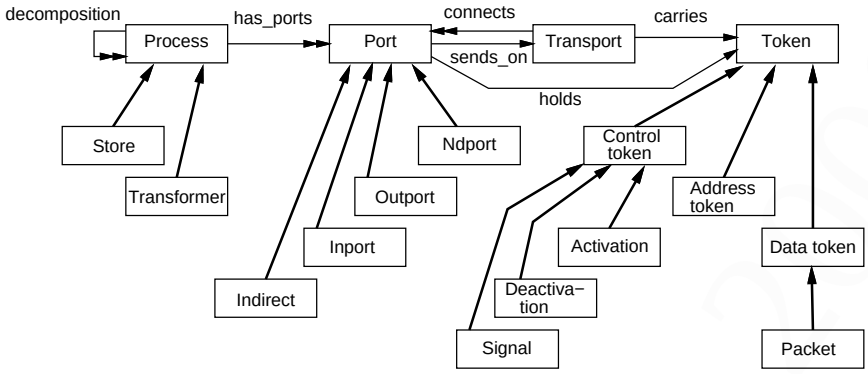


Fig. 2.39. The metamodel of GDR

2.4.4 The GDR Metamodel

GDR [245] is a design representation geared towards modeling of real-time systems, and can be used as a basis for defining languages. Examples of translations from Ward's Transformation Schema [390], from Statecharts [161], and from state transition diagrams, to GDR are given. As was the case with AMADEUS, GDR is based on a few simple, but powerful constructs. These *design objects* are organized in a class hierarchy, as shown in Fig. 2.39. Each class has a set of predefined attributes associated. In the following, we briefly describe each main class.

Processes receive information on *input ports* and produce information to be transmitted through *output ports*. Processes may either be *stores*, where inputs may be stored and later produced as outputs on requests, or *transforms*, which compute outputs from inputs. The process construct is used to represent many phenomena, including program units, objects, states, files etc. Attributes of processes describe decompositions, and link each process to its ports.

Ports identify information transmitting locations of a process. *Input ports* and *output ports* correspond to inputs and outputs of a process, while *ndports*

(non-directional) identify locations which are constrained in certain ways to other ndports. An example is when a constant relationship must be maintained between two pieces of information, e.g. for a constraint. All these ports are directly connected to *transports*, which link them to other ports. The *indirect port* is used for sending information by address, rather than through a single direct transport. Attributes of ports include associated process and transport, and the type of information which can be transmitted. In addition, output ports may transmit discrete or continuous data, while input ports may queue up incoming data or discard data when the process is inactive.

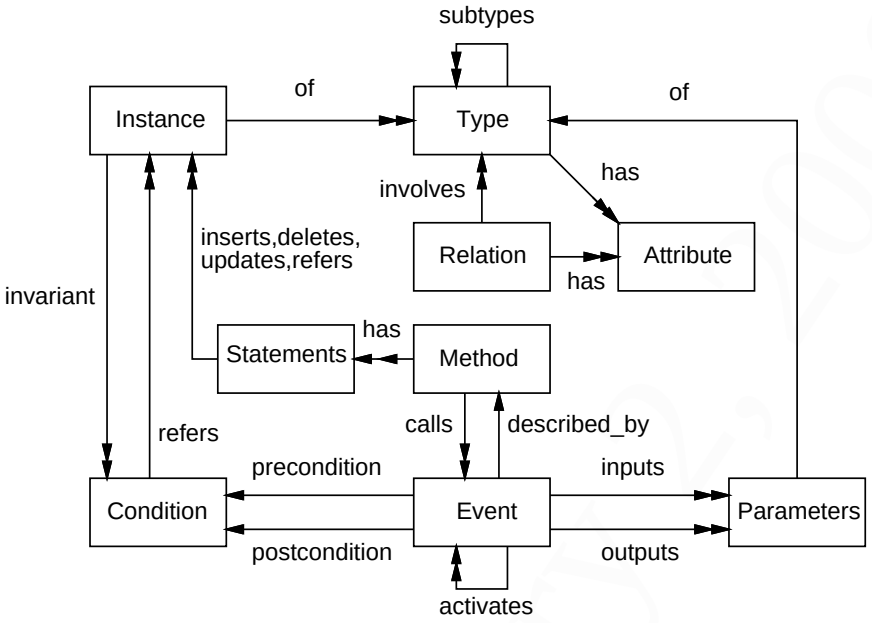


Fig. 2.40. Excerpts of the ARIES metamodel

Transports correspond to communication channels. They connect ports and facilitate communication between processes by merging and distributing information on associated ports. Attributes identify the connected ports.

Tokens correspond to various types of information. *Control tokens* can be used to activate passive processes, deactivate active processes, or to signal occurrences of events. *Data tokens* carry information used for computations. They are described by attributes which give their structure, basic types, and representation. *Address tokens* contain a port identifier, i.e. the receiver of the address token sent through an indirect port. *Packet tokens* contain an address plus data.

The semantics of GDR is to a large extent given by Petri-nets. Roughly speaking, tokens correspond to Petri-net tokens, ports correspond to Petri-net

places, and simple processes correspond to Petri-net transitions. A notable exception is data tokens, which do not have a Petri-net semantics.

2.4.5 The ARIES Metamodel

In ARIES, the intention is to provide a set of modeling languages from which developers and users can choose which one to use. The means for integration of models written in different languages is a highly expressive internal representation. Also, different presentations, both graphical and textual, can be defined from the internal representation, so that requirements can be presented in a readable manner. In ARIES, simulations of models are facilitated by translation to a database programming language described by Benner in [24].

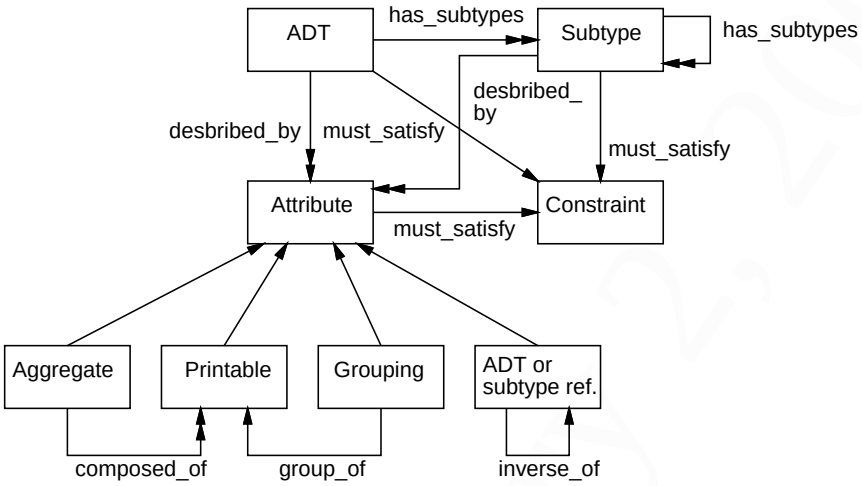


Fig. 2.41. A general semantic data model

The most important constructs of the ARIES metamodel are depicted in Fig. 2.40. The states of *instances* of *types* make up the system state. A type can have multiple subtypes and multiple super-types. This means that multiple inheritance of *attributes* is supported. Furthermore, an instance may belong to more than one type at a time. *Invariants* are used to specify constraints which must hold in all states. *Events* are used to model dynamic aspects. An event may have a *precondition* and a *postcondition*, and it may receive inputs and produce outputs through *parameters*. An event is effected through a *method* which manipulates and refers to instances. Methods use traditional control structures (sequence, iteration and choice) to control data manipulation through *statements*. An event can be activated when its precondition holds, or when it is explicitly called from within a method of another

event. Also, events may be organized into a generalization/specialization hierarchy.

2.4.6 A General Semantic Data model

Hull and King present a survey of semantic data models in [175]. From this survey one can derive a unified meta-model which covers the central constructs of newer semantic data models. This provides further insight in the requirements to expressiveness for modeling state spaces of a system. However, the meta-model focuses only on this perspective. The meta-model is given in Fig. 2.41.

The main construct is that of a *abstract data type* (ADT). Instances of an ADT belong to the *active domain* of that ADT. An ADT may have several *subtypes* (isa relationships), and instances of subtypes may be derived through a membership formula. ADTs are characterized by *attributes* which may belong to *printable types*, which means that their values can be output. Attributes may also be *aggregates* or *sets* of printable types, and their values may be derived. Relationships between ADTs are represented by attributes as well, single or multi-valued. In such cases, it can be stated that an attribute is the inverse of another. *Constraints* restrict states of ADTs and their subtypes.

Although being focused on the data perspective of conceptual modeling, this meta-model provides useful ideas not covered by the other meta-models. In particular, the use of abstraction mechanisms like aggregations and groupings has been advocated in newer semantic data models.

2.4.7 A Brief Comparison

Comparing the different meta-models, we find many similarities. On the surface, however, there may seem to be more differences than what is really the case. The differences stem from various sources. One obvious reason is different naming of similar constructs. Another reason is that sometimes, a property of a construct in one model is made explicit as a separate construct in another. Also, particular constructs may be completely lacking in one model, but exist in another. Finally, there is naturally the possibility that errors have been made in the metamodeling, since we in most cases have transformed a textual description into the graphical models.

We make a simple comparison by listing corresponding constructs in the different models, shown in Fig. 2.42. Doing this, we also highlight the three former reasons for differences between models. We will use the ontological model as a basis for comparison, since it contains a few, basic constructs, and since it has already been used for the purpose of analyzing CMLs. The constructs Environment, System, and Value&Time are omitted, since these are not found in the other models. From the table, we see that Wand and Webers

ontology, the methodology framework, and AMADEUS all have separate constructs for the data, process, and behavior perspectives. In ARIES, the event construct covers both processes and events. The unified semantic data model only covers the data perspective, while GDR emphasize more on modeling of dynamics than on modeling of data.

For representation of hierarchies in state components, the unified semantic data model offers classification, aggregation, generalization, and association. For representation of hierarchies among dynamic laws, all except the unified semantic data model has a 'vague' control structure which resemble spontaneous activation when preconditions hold. As an example, for a business activity in the methodology framework to execute, a condition must hold. In ARIES, methods include control structures from procedural programming languages.

Of the six unified meta models, only ARIES and GDR are executable. The others do not have the required detail level and a defined operational semantics.

Model	Kind	Thing	Property	State law	External event	Internal event	Transformation
Methodology framework	Relationship and entity type. Flow	Through Kind only	Attribute, attribute group, relationship	Population and value constraints. Cardinalities	Business event	Business event	Business activity
AMADEUS	Entity (in state), data flow	Data store	Entity (part–of rel.)	Facet of flow, entity,store	Event	Event	Function/ Process
GDR	Transport, ports	Store, token	Structure of data token	ndports	Token from source	Token from process	Transformer
ARIES	Type,relation	Instance	Attribute	Invariants (Condition)	Event	Event	Events with methods, cond., param., statements
Semantic datamodel	ADT, subtype	Through Kind only	Attributes of different kinds	Constraint			

Fig. 2.42. A comparison of the unified metamodels

2.5 PPP – A Multi-perspective Modeling Approach

The languages used in the PPP approach [152, 404], extended with rule-modeling as specified in [208, 214], constitute the current conceptual framework of PPP. Four interrelated modeling languages are used.

– ONER, a semantic data modeling language.
– PPM, an extension of DFD including control flows in the SA/RT-tradition.
– The rule modeling language DRL.
– AM, the actor and role modeling language.

The integrated use of the languages is supported by an experimental CASE-tool [405], where the repository structure is based on a meta-model

description of the languages, and extensibility is further supported by the use of a meta-meta-model. Modeling techniques including consistency checking, model execution, explanation generation, filtering, and code-generation to different platforms are defined and partly supported in different tools.

Below, we present the languages and their interrelationships in more detail. Examples of models made in these languages will be found throughout the rest of the book.

2.5.1 ONER – Structural and Object modeling

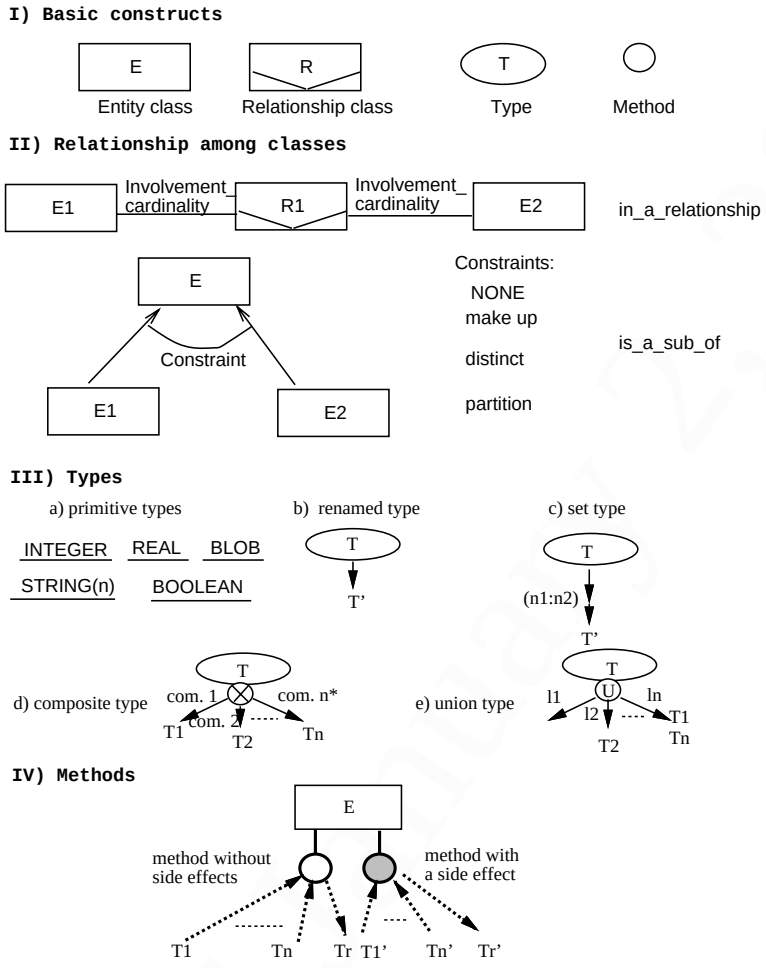


Fig. 2.43. Symbols in the ONER language

ONER is used to specify entities and relationships among entities. The symbols and vocabulary and some of the grammar rule of the ONER-language are illustrated in Fig. 2.43. To express properties of phenomena and operations on them, data types and methods are provided. Relationships can be defined between entity classes:

– *in_a_relationship*: Some members in an entity class may have relationships with members of other entity classes or with those in the same entity class. Cardinality and involvement constraints can be modeled.
– *is_a_sub_of* is a subclass relationship between two entity classes. An entity class may be a subclass of more than one class. Subclasses can make_up, be distinct within, or be a partition of the superclass.

2.5.2 PPM - Functional, Behavioral, and Communicational Modeling

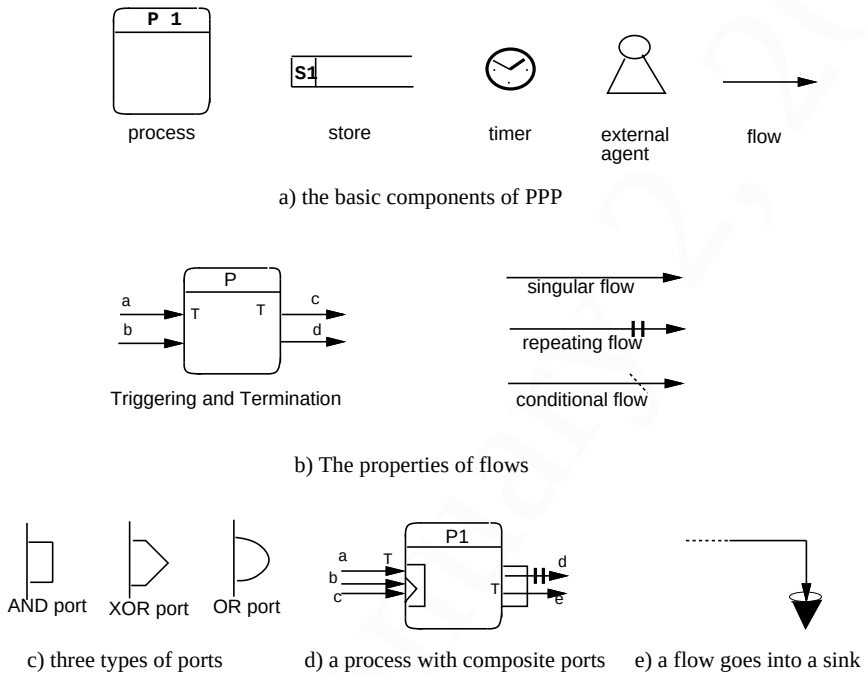


Fig. 2.44. Symbols in the PPM language

The basic modeling components of PPM are illustrated in Fig. 2.44.

– **Process:** An activity which transforms inputs to outputs.
– **Store:** A place where a collection of data or material can be kept.

- **Timer**: Clocks or delays. Clocks model events that occur at a specific time. Delays model events that are delayed a certain time interval.
- **Actor:** An actor may be a person, an organization, or a computerized information system.
- **Role:** A role is a position filled by actors. Examples of roles are 'secretary' and 'professor'.
- **Flow:** A movement of items between actors, roles, processes, stores and timers. An item has a material aspect and a data aspect; one or both aspects may be missing. In case both are missing, the item is a pure signal. When a flow appears at the input or output of a process the following properties can be specified:
  - **Triggering:** The arrival of the triggering inputs in a legal combination will start the process.
  - **Terminating:** All the termination output items will be sent out before the process changes its state to idle.
  - **Singular flow**: Only one item will be received or sent on the flow during an execution of the process.
  - **Repeating flow**: More than one item can be received or sent during execution of the process.
  - **Conditional flow**: During an execution of the process, an item may or may not be received or sent by the process.

In addition **ports** group flows showing the logical combinations of items consumed or produced by the process when executing.

- AND port: All the members of the port are going to be received or sent during an execution of the process.
- XOR port: One and only one member of the port is going to be received or sent during the execution of the process.
- OR port: At least one member of the port is going to be received or sent during the execution of the process.

This is a recursive definition, so on a process one can define *composite* port structures (Fig. 2.44d). It is possible to transform any port structure to a unique structure which is equivalent to the original port. The unique structure is of the form $\mathbf{xor}(\mathbf{and}(P_{11}, \cdots, P_{1m_1}), \cdots, \mathbf{and}(P_{n1}, \cdots, P_{nm_n}))$, where every P_{ij} is a flow. This is called a *canonical port*, which is used in connection with consistency checking. A canonical input port(CIP) is an input port that is canonical, and a canonical output port (COP) is an output port that is canonical.

Usually a flow goes from one component into another one. A possible exception is that the item in a flow will not be used i.e., the data or material is sent into the flow, then is lost. For specifying this case one has defined the auxiliary concept sink; all lost flows will be linked to sinks, as that shown in Figure 2.44*e*.

A PPM is a network of processes, stores, timers, actors, and roles connected by flows. Processes might be decomposed in additional PPMs. Items can be described by a subset of the ONER-model.

Timers are of two kinds, clock and delays:

- Clocks are used to model events that are to occur at a specific moment in time. The flows connected to a clock are of three kinds:
  - On: Starts the clock. A clock may have zero, one or many on-flows. When switched on, the clock waits a specified clock interval, issues a clock signal, waits another clock interval, and so on until it is eventually turned off. Several signals might enter through the same on-flow starting new sequences, but will not remove an old one. When no on-flow is specified, the clock is turned on when the systems is started.
  - Off: Turns the clock off. The clock may have zero, one or many off-flows. It takes a new signal on an on-flow to restart the clock again.
  - Out: The clock may have one or more outputs. All items leaving a clock will enter a process, either on a triggering or non-triggering flow. The usual situation is that it is a signal triggering a new process execution.
- Delays are used to model events that are delayed with respect to time. A delay has at least two flows connected to it, one on-flow and one out-flow. These flows can be pure signal-flows or might contain data or material. In addition, the delay may have an off-flow connected.
  When an item is received on the on-flow, it will wait the specified delay, before forwarding the item on the out-flow, if the delay is not turned of in the meantime by the reception of a similar item on the off-flow. The delay might have several on, out and off-flows.

In the case of a simple delay or clock-interval such as "2 days" (relative time) or "Friday" (explicit time), this is indicated in the graphical symbol. For most timers, this will be sufficient, but if one want a to be able to specify more complex delays and clock intervals use of rules in a when-if-then form is suggested. A timer might have several rules attached. Also external agents can be described by rules in order to simulate their behavior. How to represent most temporal relationships between processes by the use of timers is shown in [206].

Another auxiliary concept is the flowing manner that indicates how an item goes from a store. A flow going into a store will update the store in some manner. The flow from a store may consume or copy the item from the store. Copying is default, whereas a consumption is specified explicitly. In addition to this, there has been suggested to annotate PPM-models with performance parameters [286], but these will not be discussed in detail here.

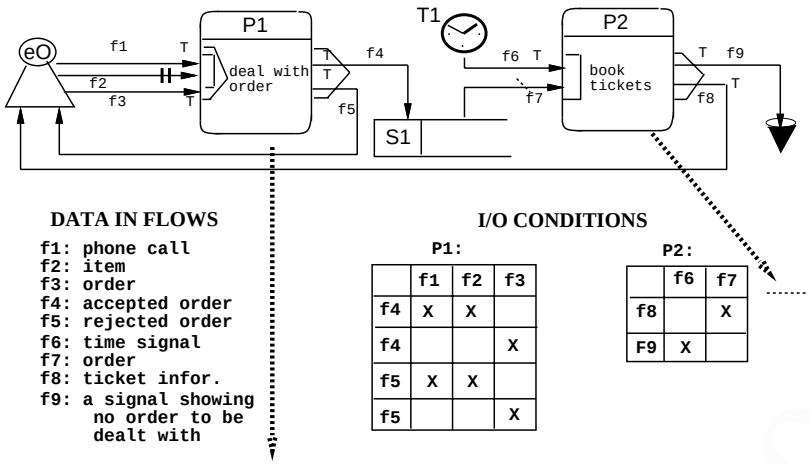
Process Logic Description. There are developed three ways of describing the process logic of a process in a PPM, as an I/O-matrix, as a PLD (Process life description), or as a set of rules similarly as in Tempora.

I/O-matrices. The relations between the input flows and the output flows of a process is the i/o condition. This is expressed by a matrix in which each column corresponds to an input and each row corresponds to an output. All rows for one output express the necessary condition for producing the output.

If an output has no row in the matrix, then it is produced unconditionally, i.e., every time when the process is executed, the output is produced.

If any output name appears in a row, then the columns on the row with "X" show a combination of inputs that may be used to produce the output. An output may occupy more than one row to indicate that more than one group of inputs may be used to produce the output.

In Fig. 2.45 we give an example which models the activities of the organization committee of the conference in connection to the ordering of tickets. Here the use of process-modeling and I/O-matrices is shown.



DATA IN FLOWS

f1: phone call
f2: item
f3: order
f4: accepted order
f5: rejected order
f6: time signal
f7: order
f8: ticket infor.
f9: a signal showing no order to be dealt with

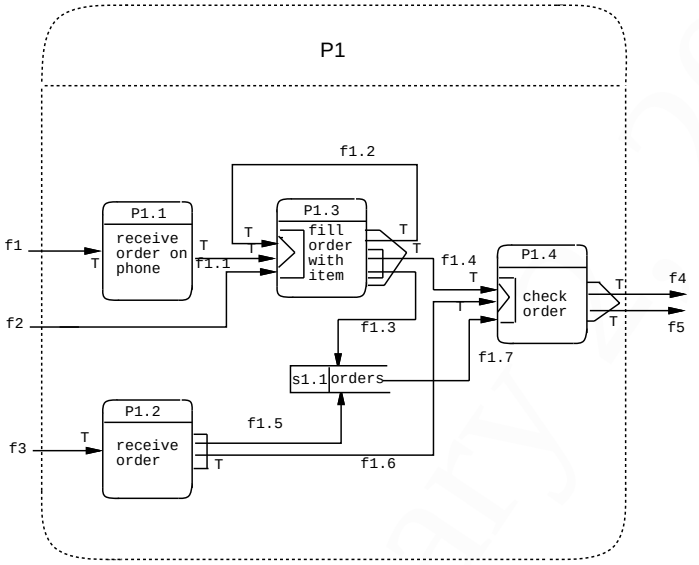
I/O CONDITIONS

P1:

	f1	f2	f3
f4	X	X	
f4			X
f5	X	X	
f5			X

P2:

	f6	f7
f8		X
F9	X	



DATA IN FLOWS

f1.1: empty order
f1.2: unfinished order
f1.3: order
f1.4: request for dealing with order
f1.5: order
f1.6: request for dealing with order
f1.7: order

I/O CONDITIONS

P1.1:

	f1
f1.1	X

P1.2:

	f3
f1.5	X
f1.6	X

P1.3:

	f1.1	f1.2	f2
f1.2	X		X
f1.2		X	X
f1.3	X		X
f1.3		X	X
f1.4	X		X
f1.4		X	X

P1.4:

	f1.4	f1.6	f1.7
f4	X		X
f4		X	X
f5	X		X
f5		X	X

Fig. 2.45. The activities for ordering tickets in the IFIP conference

PLD, process life description language. A process life description gives a procedural description of some pattern of behavior. A pattern is attachable to a process or method and can involve interactions with other patterns as well as internal computations. In this way, the description of methods can be applied directly from a PLD being part of a process.

The following elements are used to build a PLD:

- Start: Indicates the beginning of a PLD diagram.
- Receive: Is used to receive data from other PLD models.
- Assignment: Is used for variable-assignment, a PLD block or a subroutine call.
- Choice: Is used to specify selections (if and case-constructs)
- Iteration: Is used to specify loop-constructs (for and while loops).
- Send: Is used to send data to other PLD models.

Figure 2.46 shows a simple PLD model also indicating the link of the PLD to the flows in the process-models.

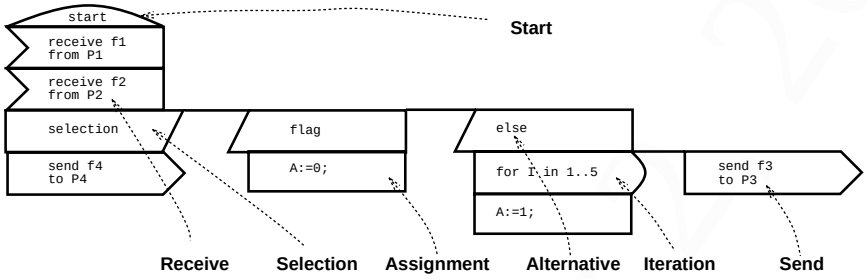


Fig. 2.46. Example of a PLD model

2.5.3 DRL - Deontic Rule Language

DRL is an extension of Tempora ERL [254]. The general rule-format is:

> when *trigger* if *condition*, it is *deontic* for *role/actor consequence* else *consequence*.

- *trigger* is optional. It refers to a state change. The trigger is expressed in a limited form of first order temporal logic.
- *condition* is an optional condition in first order temporal logic.
- *deontic* is one of the deontic operators *obligatory* (**O**), *recommended* (**R**), *permitted* (**P**), *discouraged* (**D**), and *forbidden* (**F**). A rule of necessity has no deontic operator.
- *role/actor* is an optional specification of the role or the actor that the rule applies to,

– *consequence* is an action or state which should hold given the trigger and condition.

DRL contains explicit construct for querying the ONER-model. Action rules are typically linked to processes in PPM, giving the execution semantics for the processes [212]. This is similar to how this is done in Tempora, as illustrated in Sect. 2.2.5.

Also timers and external agents can be described by rules for simulation purposes.

Both formal and informal rules are used during modeling. One can also specify necessary and deontic relationships between sets of rules, i.e. that a set of rules *necessitates*, *obligates*, *recommends*, *permits*, *discourages*, *forbids*, or *exclude* the existence of other lower-level rules [208]. Exception-hierarchies through *overrules* and *suspends* relationships can also be expressed.

In addition to the relationships, the following mechanisms are used to create a directed acyclic graphs (DAGs) with and/or-nodes of rules. The description below refers to Fig. 2.47.

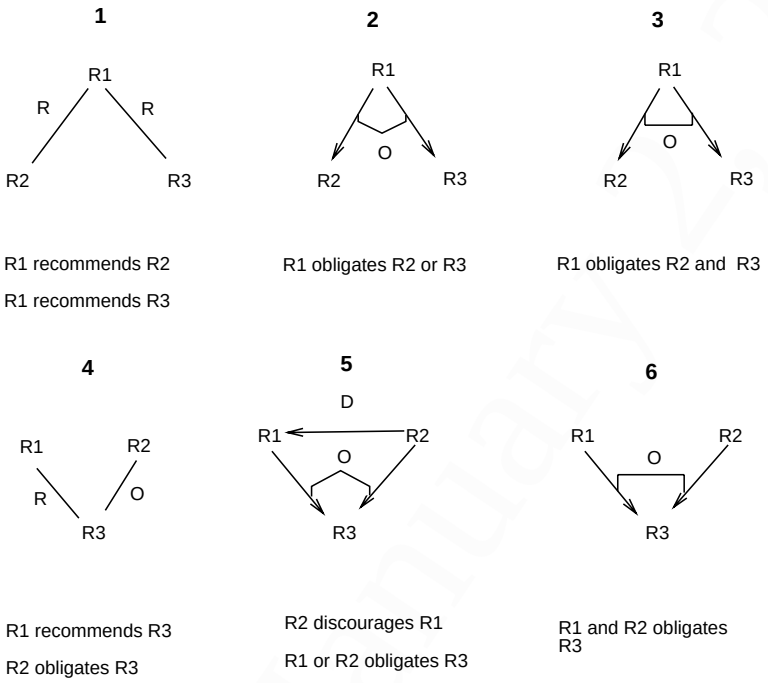


Fig. 2.47. Rule-hierarchies

1. The usual case with rule **R1** recommends rule **R2** and **R3** independently.

2. Illustrates an or-situation, thus similar situations as described by Easterbrook in Sect. 2.2.5 are supported. Based on rule **R1**, there are two perceived ways of partly fulfilling this in a lower level rule. These two rules, **R2** and **R3** might be inconsistent, even if this is not indicated in the diagram.
3. This is similar to the first situation, but indicates that both **R2** and **R3** should be fulfilled simultaneously if one wants to fulfill **R1**, thus **R2** and **R3** are subgoals of **R1**. This is parallel to the *subgoalof* relationship proposed by Feather. This approach is often used in projects using military standards, see e.g. [73, 11].
4. Indicates that rule **R3** is both recommended by rule **R1** and obligated by rule **R2**. These higher level rules can have the same source, or they represent the views of two different actors.
5. Indicates that rule **R3** is obligated by either rule **R1** or rule **R2**, but not both, since in this case, having rule **R2** in place discourages having rule **R1**.
6. Similar to the fourth situation, but indicates that it is the simultaneous fulfilling of **R1** and **R2** that obligates **R3**. The reason for modeling this kind of situation is that one might choose **R1** not to be valid of some reason. With this situation, **R3** would not be regarded as obligated.

Both functional and non-functional and specific project goals and rules can be part of the goal-hierarchy, although we do not propose an automatic support of NFR goals as in [270]. This could be a part of the general model of the developer. When to introduce functional and non-functional goal should be taken care of in the methodology assuring that the approach still will be problem-oriented and not product-oriented to the extent possible.

2.5.4 AML – Actor Modeling Language

The main symbols used for the modeling of actors are shown in Fig. 2.48.

Actors and roles are connected in the following ways:

– Relationships: Four general relationships are defined:
1. An actor can be *part of* another (organizational) actor. One actor can be part of many actors at the same time.
2. An actor can (be expected to) fill a role. One actor can fill more than one role, and a role can be filled by more than one actor. A role can be identified as a set of expectations which normally also will apply to the actors that fill the role.
3. A role can be *part of* another role, i.e. some of the expectations of a role also applies to another role.
4. A role is instituted by an actor. The set of expectations to a role often comes from one or more actors. If there are specific expectations to an actor in a given role, this can be depicted using the agent-symbol.

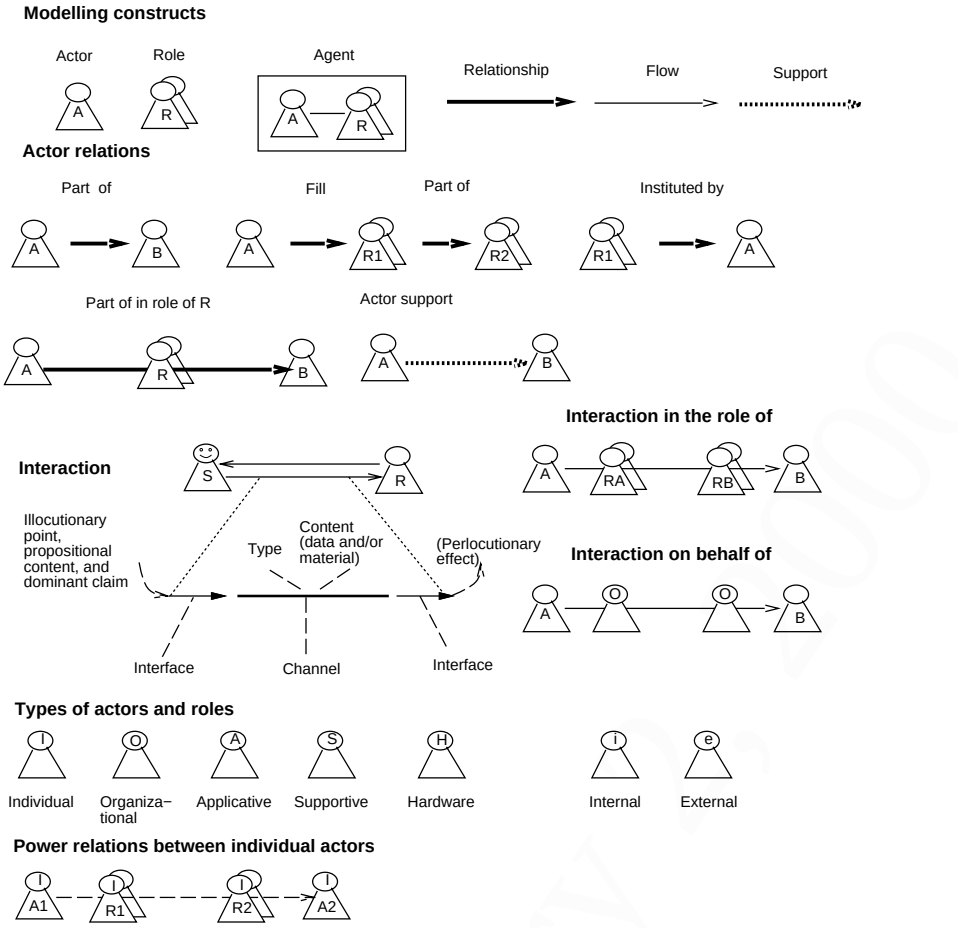


Fig. 2.48. Basis for actor models

– Support: Actors and roles can support other actors and roles in performing their tasks. We will indicate what this means in more detail when discussing types of actors and roles below. It is also possible to indicate that actors *potentially* support other actors.
– Communicate: Actors and roles that are filled by an actor can communicate by exchanging items over flows.

Roles can also be related with over-rules relationships, indicating in the cases of role-conflict which role that is regarded as most important, other things being equal. One can also indicate for which actor this applies if necessary.

The interaction between actors can be divided up further, much in line with the interprocess-communication in PPM, but with certain possibilities

for extension when it comes to language action modeling. In addition to the type and contents of the items being sent, how many that they are sent to or received by, and if they trigger the other actor, one can indicate the set of illocutionary points, propositional contents, and dominant claims of the communication.

Relationships between actors can be further annotated, by additional actors or roles on both sides of the relationship.

- *In the role of*: An actor can support, communicate, or be part of another actor in a given role. In the cases of communicate and support one can also annotate the receiving actor similarly.
- *On behalf of*: An actor can communicate, support, or be part of another actor on behalf of another actor. In the cases of communicate and support one can also annotate the receiving actor similarly.

It is distinguished between the following types of actors:

- Individual actor (I) e.g. you and me.
- Organizational actor (E) e.g. IDI, the institute.
- Applicative software actor (A) e.g. the conference system C4.
- Supportive software actor (S) e.g. Ingres DBMS.
- Hardware actor (H) e.g. the file server storlind.

One can also indicate if an actor is external (e) or internal (i) relative to some organizational actor. The internal actor is typically the organization in which the CIS is being developed and maintained, but do not need to, i.e. one can easily produce different views of actor-models.

The types of actors above can freely be mixed with the internal/external classification. Roles can also have the types as indicated above.

The term *support* has the following more specialized meanings.

- If one hardware-actor support another hardware-actor it is used by it to achieve an overall goal that some social actor has set for it. General hardware compatibility is indicated through support.
- If one hardware actor can support a software actor (supportive or applicative), the software actor is executed on it. General executional compatibility is indicated through a potential support.
- If one software actor supports another software actor, it is used by it to achieve an overall goal that some social actor has set for it. General software compatibility is indicated by potential support.
- If a computational actor supports a social actor, it means that it is used by the individual or organizational actor to achieve further goals.
- If a social actor support a computational actor, then they ensure the possibility for the computational actor to support additional organizational actors. Thus these actors indirectly support the other organizational actors.

– That a social actor support another social actor indicate that it helps the other actor to achieve its goal in some way.

Finally, it is possible to indicate the power-relationships between individual actors and roles, indicating the power structure in an organization.

2.5.5 UID – User Interface Description Language

Although the model of the user interface is usually not regarded as a conceptual model, we will briefly describe the languages for developing such models in PPP, including the links to the other languages, since the user-interface is an important part of most application systems. The description is based on [111, 310].

The model separate the presentation and behavioral parts of a user interface. By the presentation part one means the part of the interface which is visible to the user. By the behavioral part one refers to the interaction between objects of the presentation part, and interaction with other parts of the application. The two parts of UID are called UIP (User Interface Presentation) and UDD (User interface Dialog Description), respectively.

Presentation – UIP . The language to represent the presentation part of the user interface is object oriented.

– Components of a user interface can be modeled as encapsulated objects with a defined set of methods and interfaces. Execution of an application implies interaction between these objects and other parts of the application by means of message passing.
– Components of a user interface can be grouped in families of objects with similar behavior. UIP makes it possible to generate user defined classes of interface components.

Each class in the user-interface hierarchy have a set of properties that can have values, a set of methods that can change or retrieve these values, and a set of events that an object of the class can react to. A state of a screen in a user-interface is defined as a mapping of values to all properties except value contents of all objects in the screen. When one of these changes due to an event, a state transition takes place.

Behavior – UDD. Interaction between components of a user interface and its environment is modeled by UDD. A user interface will be a set of screens which can be thought of as a set of state machines as described above. Transitions are defined by the set of services available in each state and the environment the interface is used. UDD is based on Statecharts (see Sect. 2.2). One Statechart is made for each screen.

The extensions in Fig. 2.49 are made to Statecharts in UDD.

– Events not causing transitions: Events which alter data attributes often do not cause state transitions. Such an event is presented by a special symbol within the state. This possibility is also included in OMT.

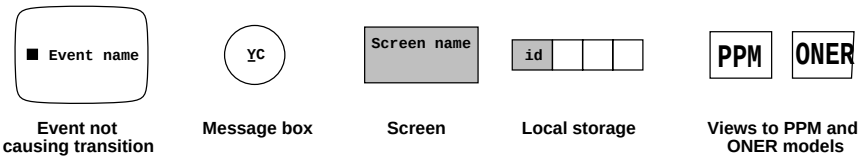


Fig. 2.49. Symbols in the extensions to Statecharts used in UDD

– Message box: A typical graphical user interface uses message boxes extensively. To simplify the diagram of such user interfaces, explicit message boxes are used in UID.
– Screen: When modeling a screen, one often needs to include other screens in the diagram because of possible transition of control out of the current screen and into another.
– Local Storage Structure: Data structures to store data for the screen which are not shown to the user.
– View: Connections to PPM and ONER models.

Often one will need to perform a selection of data from the database when invoking a screen, and this is specified with a scenario including selection criteria (using the defined algebra on ONER models). The PPM view is used to indicate the action of an event, thus indicating which data in the user interface (including the local structure) which will be sent to trigger a process. This will be linked to one or more outputs from processes "further down" in the PPM, to indicate when to return control to the user-interface, and where to put outputs.

2.6 Chapter Summary

We have in this chapter looked into two main aspects of conceptual modeling languages. The use of abstraction mechanisms, and the use of perspectives.

A conceptual model is an abstraction. One mechanism for abstraction used in many of the existing languages for conceptual modeling is the use of *hierarchies*. There is a vast number of hierarchies that one might want to model, and these have rather diverse properties. Work in the field of semantic data modeling and semantic networks has lead to the identification of four standard hierarchical relations:

– **classification**,
– **aggregation**,
– **association**, and
– **generalization**.

Modeling languages can be divided into classes according to the core phenomena classes that are represented in the language. We have called this the main *perspective* of the language.

To give a broad overview of the different perspectives state-of-the-art conceptual modeling approaches accommodate, we have described languages taking the following perspectives:

- Structural perspective. Examples of languages are ER, ONER, GSM, ERT, semantic networks, and conceptual graphs.
- Functional perspective. Examples of languages are DFD, PID, PPM, SA/RT and the work by Sindre and Opdahl.
- Behavioral perspective. Example of languages are STD, STM, Statecharts, Petri-Nets, and BNM.
- Rule perspective. Examples of languages are COMEX, ERL, DRL; and the work of Mylopoulos and Sutcliffe.
- Object perspective. Examples of languages are OMT, UML, HOOD, and OOA.
- Communication perspective. Examples of languages are Action Workflow, SAMPO, ABC/DEMO, and COMMODIOUS.
- Actor and role perspective. Examples of languages are AM, ALBERT, OORASS, and the work of Mylopoulos et al.

A model in a given language having only one perspective will seldom be sufficient to capture all interesting aspects of a situation. With this in mind more and more approaches are based on the combination of several modeling perspectives. Four ways of approaching this are:

1. Use existing single-perspective languages as they are defined, without trying to integrate them further.
2. Refine common approaches to make a set of formally integrated, but still partly independent set of languages.
3. Develop a set of entirely new integrated conceptual modeling languages.
4. Create frameworks that can be used for creating the modeling languages that are deemed necessary in any given situation.

A consequence of a combined approach is that it requires a much better tool support to be practical. Due to the increased possibilities of consistency checking and traceability across models, in addition to better possibilities for the conceptual models to serve as input for code-generation, and to support validation techniques such as execution, explanation generation, and animation, the second of these approaches has been receiving increased interest. Basing the modeling languages on well-known modeling languages also have other advantages on behalf of perceptibility, and because of the existing practical experience with these languages. Work based on language-modeling might also be used to improve the applicability of the other approaches. PPP is one such approach that has been presented in more detail.

3. Quality of Conceptual Models

In this chapter we describe a framework for understanding quality in conceptual modeling. 'Quality' is a difficult notion, and within the field of information systems, many approaches to quality have been proposed. A standard approach within the engineering community is to say that a product has high quality if it is according to its specification. For example the ISO 9000 quality standard is set up according to this philosophy. Denning [89] takes this further by viewing this as the first level of quality. A second level is achieved if there are no negative side-effects of the information system. The highest level of quality is achieved if in addition to the first two levels, the information system enables additional information system support to its users not conceived in the first place, i.e. giving the users more of what they need than what was promised in the specification.

Previous proposals for quality goals for conceptual models and requirement specifications as summarized by Davis [81] have included many useful aspects, but unfortunately in the form of unsystematic lists as discussed in [236, 239]. They are also often restricted in the kind of models they regard (e.g. requirements specifications [73, 81]) or the modeling language (e.g. ER-models [263]). Some recent frameworks [109, 239, 306] have attempted to take a more structured approach to understand quality, and the framework presented here integrates aspects from all of these.

3.1 Overview and Evaluation of Existing Frameworks

We will here give a short overview and comparison of three existing frameworks:

- Lindland et al [239].
- NATURE [306].
- FRISCO [109].

Whereas [239] was briefly described in Chap. 1, we will here describe the other two framework briefly, together with some of the motivation behind the combined framework.

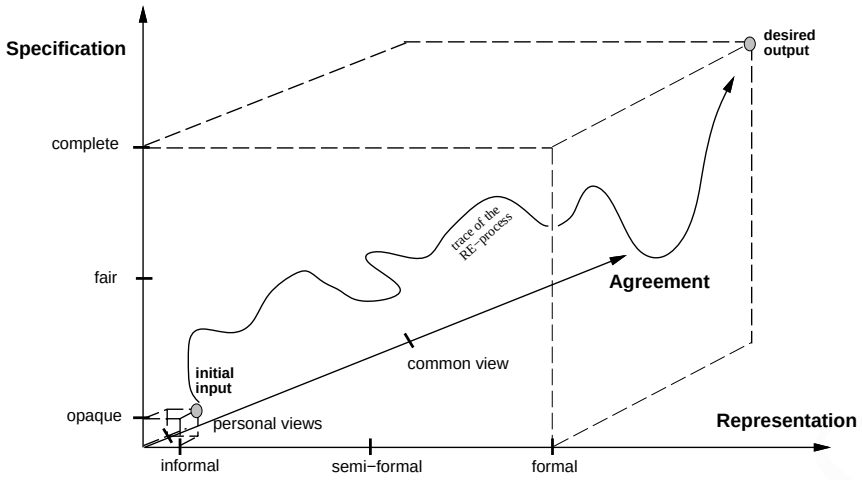


Fig. 3.1. Pohl's framework (From [306])

3.1.1 Pohl's Framework

Pohl's framework [306] which is part of the NATURE-project [186], is summarized in Fig. 3.1. In this framework the requirements specification process which often includes conceptual modeling, is stretched out along three dimensions:

- *the specification dimension* deals with the degree of requirements understanding. At the beginning of the process, this understanding is opaque. The desired output of the requirement specification process is a complete CIS-specification, where completeness is measured against some standard, guideline, or some other model.
- *the representation dimension* deals with the degree of formality. At the beginning of the process, statements will usually be informal. Since formal representations allow reasoning and partial code-generation, these are more product-oriented. Hence, a transformation of informal requirements to a formal representation is regarded as desirable.
- *the agreement dimension* deals with the degree of agreement. The requirement specification process has many stakeholders, and in the beginning each of these will have their personal views concerning the requirements. The goal of the process is to reach agreement on these requirements. Detected conflicts must be resolved through discussions among those affected.

3.1.2 FRISCO

The FRISCO report [109] identifies that the means of communication and related areas can be examined in a *semiotic* framework. The below semiotic

layers for communication are distinguished, forming a semiotic ladder. Model denotations are sign, and thus they have considered the semiotics of models. The key concepts to be included in information systems models is regarded to be

- *Physical:* use of various media for modeling - documents, wall charts, computer-based CASE-tolls and so on; physical size and amount and effort to manipulate them; human resources needed; economics.
- *Empirical:* variety of elements distinguished; error frequencies when being written and read by different users; coding (shapes of boxes); ergonomics of computer-human interaction (CHI) for documentation and CASE tools.
- *Syntactic:* languages, natural, constrained or formal, logical and mathematical methods for modeling.
- *Semantic:* interpretation of the elements of the model in terms of the real world; ontological assumptions; operations for arriving at values of elements; justification of external validity.
- *Pragmatic:* roles played by models - hypothesis, directive, description, expectation; responsibility for making and using the model; conversations needed to develop and use the model.
- *Social:* communities of users; the norms governing use for different purposes; organizational framework for using the model.

These lists are indicative rather than exhaustive. These layers can be divided into two groups in order to reveal the technical vs. the social aspect. Physics, empirics, and syntactics comprise an area where technical and formal methods are adequate. However, semantics, pragmatics, and the social sphere cannot be explored using those methods unmodified. This indicates than one has to include human judgment when discussing concepts in the higher semiotic levels.

3.1.3 Overall Comparison

Although the frameworks of Lindland et al. and Pohl have quite different appearances, they are rather similar in their deeper structure. The following observations can be made:

- the *representation* dimension corresponds to the *syntactic* dimension, since both these deal with the relationship between the specification and the language(s) used. The main differences in this respect is that Pohl's framework discusses several languages, whereas Lindland et al.'s framework sees the language as one and just considers whether the specification is correct according to the rules of that language (which may be a union of several languages, formal, semi-formal, and informal). It should also be noted that Pohl's framework regards a formal specification as a *goal*. Lindland's framework states that formality is a *mean* to reach a syntactically correct specification, as well as higher semantic and pragmatic quality through consistency checking and model executions of different kinds.

- the *specification* dimension corresponds to the *semantic* dimension, since both these deal with the goal of completeness. A notable difference here is that Pohl sees completeness as the sole goal (possibly including validity?), whereas Lindland's framework also identifies the notions of validity and feasibility. The reason for this discrepancy seems to be a somewhat different use of the term completeness. Pohl uses the term relative to some standard, whereas Lindland et al. uses it relative to the set of all statements which would be correct and relevant about the problem at hand.
- the *agreement* dimension is related to the *pragmatic* dimension, since both these deal with the specification's relationship to the audience involved. The difference is that Pohl states the goal that the specification should be agreed upon, whereas Lindland et al. aim at letting the model be understood. In a way these goals are partly overlapping. Agreement without understanding is not very useful in a democratic process. On the other hand, using the semiotic levels of FRISCO, it is more appropriate to put agreement into the social realm going beyond the framework of Lindland et al.

Comparing the Lindland framework to FRISCO, we see that the framework suggested by Lindland et al. to some extend take the insight of semiotic levels into account by differentiating between syntactic, semantic, and pragmatic quality. Even if the terms are used somewhat differently, the overall levels coincide. On the other hand, neither the lower physical and empirical level or the social level can be said to be discussed and covered in the existing framework. As indicated above, the social aspects of agreement is currently not handled in a satisfactory way. When discussing agreement, the concept 'domain', as currently defined, is also problematic, since it represents an ideal knowledge about a particular situation, which is not obtainable for the actors of the audience that are to agree.

Based on this, we present an extension of the framework of Lindland et al. taking the above aspects into account. Parts of this extended framework have earlier been presented in [207, 210, 211, 345]. In addition we include a discussion of language quality based on [331, 344]. Language quality together with knowledge quality allows us to focus in more detail on the aspects which in the original framework were termed rather loosely 'appropriateness'.

3.2 A Framework for Quality of Conceptual Models

In this section, we outline the overall framework. The framework has three unique properties:

- It distinguishes between goals and means by separating what you are trying to achieve from how to achieve it.
- It is closely linked to linguistic and semiotic concepts. In particular, the core of the framework including the discussion on syntax, semantics, and

pragmatics is parallel to the use of these terms in the semiotic theory of Morris (see e.g. [276] for an introduction). The inclusion of such semiotic levels enables us to address quality at different levels. A term such a 'quality' is used on all the semiotic levels. We include physical, empirical, syntactical, semantical, pragmatic, and social quality in addition to knowledge and language quality. A brief overview of each area is given in this chapter. Means for achieving syntactical, semantical, pragmatic, and social quality is discussed in more detail in separate chapters.

– It is based on a constructivistic world-view, recognizing that models are usually created as part of a dialogue between the participants involved in modeling, whose knowledge of the modeling domain changes as modeling takes place.

The main concepts and their relationships are shown in Fig. 3.2. We take a set-theoretic approach to the discussion of the different quality aspects. Sets are written using $\mathcal{CALIGRAPHIC}$ letters, whereas elements of sets are written in normal uppercase letters. An overview of the symbols used can be found in Appendix C. Readers familiar with the field of logic programming should be aware of that we use several terms differently from how they are used in that field.

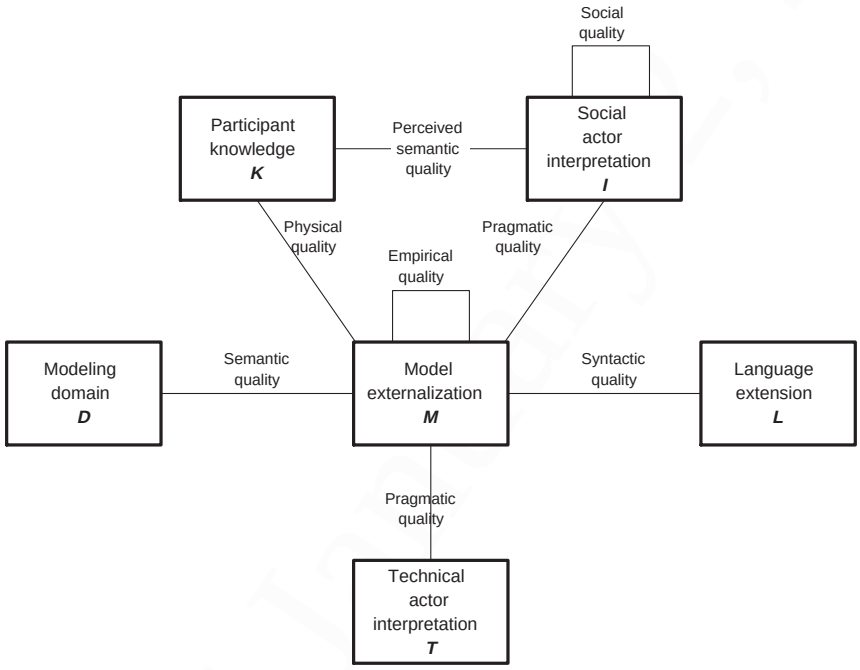


Fig. 3.2. Extended framework for discussing quality of conceptual models

Important sets are:

- $\mathcal{A}$, the audience, i.e. the union of the set of individual actors A_1,...,A_k the set of organizational actors A_{k+1},...,A_n and the set of technical actors A_{n+1},...,A_m who need to relate to the model. The individuals being members of the audience are called the *participants* of the modeling process. The participants $\mathcal{P}$ is a subset of the set of stakeholders $\mathcal{S}$ of the process of creating the model. In general, stakeholders of a system development effort can be divided into the following groups [250]:
  - Those who are responsible for design, development, introduction, and maintenance of the CIS, for example, the project manager, system developers, communications experts, technical authors, training and user support staff, and their managers.
  - Those with financial interest, e.g. those responsible for the application systems sale or purchase.
  - Those who have an interest in its use, for example direct or indirect users and users managers.
  - Those who are to support the users on technical matters on a day to day basis.

  A technical actor is typically a computer program e.g. a CASE-tool, which must "understand" parts of the model to automatically manipulate it to for instance perform code-generation or to execute the conceptual model.
  The audience often changes during the process of developing the model, when people leave or enter the project.
- $\mathcal{L}$, the language extension, i.e. the set of all statements that are possible to make according to the vocabulary and syntax of the modeling languages used. Several languages can be used in the same modeling effort, corresponding to the sets $\mathcal{L}_1$,...,$\mathcal{L}_j$. These languages can be inter-related. Sub-languages are related to the complete language by limitations on the vocabulary or on the set of allowed grammar rules in the syntax of the overall language or both. The statements in the language model of a formal or semi-formal language L_i are denoted with $\mathcal{M}(L_i)$. This model is often called the meta-model of the language, a term which is appropriate only in connection to work on repositories for conceptual models.
  $\mathcal{L}$ can be divided into $\mathcal{L}_I$, $\mathcal{L}_S$, and $\mathcal{L}_F$ for statements made in informal, semi-formal and formal parts of the language, respectively. $\mathcal{L} = \mathcal{L}_I \cup \mathcal{L}_S \cup \mathcal{L}_F$. $\mathcal{L}_L$ denotes the statements with logical semantics.
  The languages used in a modeling effort are often predefined, but it also happens that one create specific modeling languages using e.g. a meta-CASE tool for the modeling effort, in which case the syntax and semantics of the languages have to be inter-subjectively agreed among the audience as part of modeling. If one are using an existing language, the "correct" syntax and semantics of the language will be regarded as predefined. One can choose to apply only parts of the predefined modeling languages for a given

modeling effort, and change this subset during a project as appropriate. We will look into how this can be done in practice in Chap. 8.

- $\mathcal{M}$, the externalized model, i.e. the set of all statements in someones model of part of the perceived reality written in a language. $\mathcal{M}_E$ is the set of explicit statements in a model, whereas $\mathcal{M}_I$ is the set of implicit statements, being the statements not made, but implied through the deduction rules of the modeling language. For example, assume that L is propositional logic and $\mathcal{M}_E$ contains the statement $A \rightarrow B$ and A. $\mathcal{M}_I$ will contain the derived statement B. A model written in language L_i is written $\mathcal{M}_{L_i}$. The meaning of $\mathcal{M}_{L_i}$ is established through the inter-subjectively agreed syntax and semantics of L_i.
  For each participant, the part of the externalized model which is considered relevant can be seen as a projection of the total externalized model, hence $\mathcal{M}$ can be divided into projections $\mathcal{M}^1, ..., \mathcal{M}^k$ corresponding to the participants $A_1, ..., A_k$. Generally, these projections will not be disjoint, but the union of the projections should cover $\mathcal{M}$. $\mathcal{M}$ will obviously evolve during modeling as statements are inserted and deleted into the model.
  Another important distinction, is between the model as accessed by the users, and the internal representations of the model which the technical actors have to relate to.
- $\mathcal{D}$, the modeling domain, i.e. the set of all statements which can be stated about the situation at hand. If one use an objectivistic ontology, or one accept a high degree of inter-subjective agreement on the modeling domain, this is similar to the definition of the original framework.
  One can differentiate between domains along two orthogonal dimensions:
  - Temporal: Is the model of a past, current, or future situation as it is perceived by someone in the audience?
  - Scope: Examples of different scopes are: (A subset of) the physical world, (a subset of) the social world, an organization, an information system, a computerized information system.

  More specifically, during development of a CIS, several different although interrelated modeling domains, with accompanying models are recognised [177]:
  - The existing IS as it is perceived, $\mathcal{M}(EIS)$. Another description of this is the internalization of the current organizational reality.
  - A future IS as it is perceived, e.g. requirements to a future IS, $\mathcal{M}(FIS)$.
  - The external behavior of the future CIS as it is perceived, e.g. requirements to a future CIS. This can be regarded as an extension of $\mathcal{M}(FIS)$.
  - The internal behavior of the future CIS as it is perceived, e.g. design of a future CIS, $\mathcal{M}(FCIS)$.
  - The implemented CIS. Also the CIS can be regarded as a model, although usually not a conceptual model in the sense we use the term [36]. Specifically when a CIS is populated with dynamically changing data,

one can use the quality framework to look at aspects regarding areas such as data quality. This is not the focus of this book.

The domains evolves during modeling, both because of the modeling itself and external changes. Any of the above domains can be subdivided into three parts, exemplified by looking at a software requirements specification [81]:

- Everything the CIS is supposed to do (for the moment ignoring the different views the stakeholders have on the CIS to be produced). This is termed the primary domain.
- Constraints on the model because of earlier baselined models such as system level requirements specifications, statements of work, and earlier versions of the requirement specification to which the new requirement specification must be compatible. This is termed the pre-existing context.
- Constraints through the fact that one wants to produce CIS based on the software requirement specifications under given time and resource limits. This is termed the purpose context.

- $\mathcal{K}$, the relevant explicit knowledge of the audience, i.e. the union of the set of statements, $\mathcal{K}_1$,...,$\mathcal{K}_k$, one for each participant. $\mathcal{K}_i$ is all possible statements that would be correct and relevant for addressing the problem at hand according to the explicit knowledge of participant A_i. $\mathcal{K}_i \subset \mathcal{K}^i$, the explicit internal reality of the social actor A_i. $\mathcal{M}_i$ is an externalization of $\mathcal{K}_i$ and is a model made on the basis of the knowledge of the individual or organizational actor. Even if the internal reality of each individual will always differ, the explicit internal reality concerning a constrained area might be equal, especially within specific groups of participants [136, 291]. Thus it can be meaningful to also speak about the explicit knowledge of an organizational actor. $\mathcal{M}_i \setminus \mathcal{M}^i = \emptyset$, whereas the opposite might not be true, i.e. more of the total externalized model than the part which is an externalization of parts of an actor's internal reality is potentially relevant for this actor. $\mathcal{K}$ will and should change during modeling to achieve both personal and organizational learning [380].

  Representing knowledge as sets of statements is not to claim that this is how knowledge is actually stored in the human brain, for this claim would clash with advances in brain sciences on this topic [65]. On the other hand, it is a useful abstraction for the kind of knowledge it is possible to represent explicitly using a language.
- $\mathcal{I}$, the social audience interpretation, i.e. the set of all statements which the social audience perceive that an externalized model consists of. Just like for the externalized model itself, its interpretation can be projected into $\mathcal{I}_1, ..., \mathcal{I}_n$ denoting the statements in the externalized model that are perceived by each social actor.
- $\mathcal{T}$, the technical audience interpretation. Similar to above $\mathcal{I}_{n+1}, ..., \mathcal{I}_m$ denote the statements in the conceptual model as they are interpreted by each technical actor in the audience.

The primary goal for semantic quality is a correspondence between the externalized model and the domain, but this correspondence can neither be established nor checked directly: to build the model, one has to go through the participants' knowledge regarding the domain, and to check the model one has to compare this with the participants' interpretation of the externalized model. Hence, what we observe at quality control is not the actual semantic quality of the model, but a *perceived semantic quality* based on comparisons of the two imperfect interpretations.

Table 3.1 shows an overview of the goals and means that have been identified on the different semiotic levels. The means for syntactic, semantic, pragmatic, and social quality will be discussed further in Chapters 4-7. Language quality goals are looked upon as means in the framework. Language quality is discussed briefly towards the end of this chapter. A specific technique is positioned as a mean in the category where it is believed to be of most importance. We have also indicated quality types that can be beneficial to achieve before attacking the relevant area, thus indirectly the means for achieving e.g. empirical quality is also a potential mean for achieving pragmatic quality.

Table 3.1. Framework for model quality

Quality type	*Goals*	*Means* Beneficial existing quality	Model and language properties	Activities/ Tool-support
Physical	Externalization		Domain app. Part. knowledge app.	Meta-model adaption
	Internalizability		Persistence Availability	DB-activities/ repository
Empirical	Minimal error frequency	Physical (externalization)	Expressive economy Aesthetics	Readability index Diagram layout
Syntactic	Syntactic correctness	Physical (externalization)	Formal syntax	Error prevention Error detection Error correction
Semantic	Feas. validity Feas. completeness	Physical (externalization) Syntactic	Formal semantics Modifiability	Consistency checking Statement insertion Statement deletion Driving questions Model reuse Model testing
Pragmatic	Feasible comprehension	Physical Empirical Syntactic	Operational semantics	Inspection Visualisation Filtering Rephrasing Paraphrasing Explanation
			Executability	Execution Animation Simulation
Perceived semantic	Perc. validity Perc. completeness	Physical Empirical Syntactic Pragmatic	Variety	Participant training
Social	Feasible agreement	Physical Pragmatic Perc. Semantic	Inconsistency handling	Model integration Conflict resolution
Knowledge	Feas. knowledge completeness Feas.knowledge validity			Stakeholder identification Participant selection Problem selection

Throughout the chapter we will use part of an ER-model from the conference-case as depicted in Fig. 3.3 to illustrate the different aspects. The numbers on the figure refer to different statements in the model.

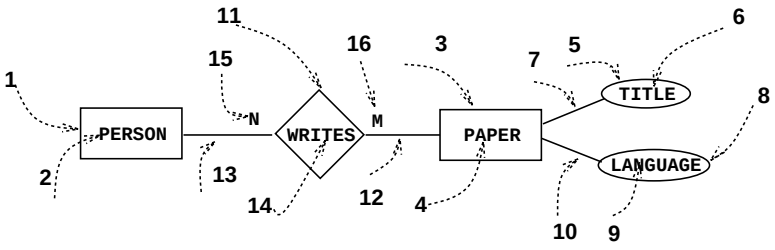


Fig. 3.3. A simple ER-diagram

3.3 Physical Quality

Although information system models are not usually of the physical kind, any model can be represented physically somehow, e.g. on disk or paper. In our example, it is basically represented on paper as Fig. 3.3. The basic quality features on the physical level is externalization, that the knowledge of some social actor has been externalized by the use of a conceptual modeling language, and internalizability, that the externalized model is persistent and available enabling the audience to make sense of it. This is not the same as internalization of the model.

Externalization can be defined as the number of statements known about the domain that is not yet stated in the model divided by this number of statements. Mathematically, this can be stated:

$$externalization = 1 - \frac{\#(\mathcal{K} \backslash \mathcal{M})}{\#(\mathcal{K})} \tag{3.1}$$

The major means for achieving this are using modeling languages which are appropriate for the domain and participant knowledge. This will be discussed further under language quality.

Internalizability on the physical level has two primary means, persistence and availability:

– Persistence: How persistent is the model, how protected is it against loss or damage? This also includes previous versions of the model, if these are relevant. A previous version of the model will be part of the modeling domain and it might be necessary to model the relationship to this.

– Availability: How available is the model to the audience? Clearly, this is dependent on its externalization. Availability also depends on distributability, especially if members of the audience are geographically dispersed. Then, a model which is in an electronically distributable format will be more easily ditributed than one which must be printed on paper and sent by ordinary mail or fax. It may also matter exactly what is distributed, e.g. the model in an editable form or merely in an output format.

One important activity in this area is the adaption of the meta-model of the basic language used to suit the domain, both by adding concepts, but also by removing concepts (temporarily) from the language if they are not relevant for the modeling of the particular domain. This is treated in more detail under the discussion on domain appropriateness in Sect. 3.11. Many of the other activities in connection with physical quality are typically based on traditional database-functionality using a repository-solution for the internal representation of the model [168, 201]. In addition,it is regarded necessary for advanced tools for conceptual modeling and system development to include functionality such as version control and configuration management and advanced concurrency control mechanism, that are not normally found in conventional DBMSs [168]. A more detailed list of tool-mechanisms, most of them concerning availability, is presented below:

– Data independence: Modeling tools do not need to change when physical storage is rearranged, so long as the logical schema remains unaltered.
– Openness: New tools with access to the same model are easily added.
– Queries: The DBMS provides a simple user interface for interactive queries.
– Reports: Users can easily define standard reports.
– Real-time updating: The latest state of the model is always available. Due to the long transactions often needed to make consistent new versions of a model, it is doubtful that one would like the same kind of real-time update as in a transaction system. On the other hand, when a new consistent version is available, it should be available for the audience as quickly as possible.
– Locking: Simultaneous attempts to update the same object are prevented.
– Concurrency: Subject to the bar on simultaneous updates, multiple tools and multiple users may work concurrently on "live" data.
– Security: Objects may be protected from unauthorized reading, writing, or deletion.
– Transactions: Being able to store a set of changes to the model as a whole, or not at all.
– Recovery: After any abnormal event, the model is restored to a truthful and, as far as possible, up-to-date state.
– High performance.
– Versioning and configuration management.

Aspects related to versioning of conceptual models and concurrency control will be discussed in Chap. 8.

3.4 Empirical quality

As indicated above, empirical quality deals with the variety of elements distinguished, error frequencies when being written or read, coding (shapes of boxes) and ergonomics for Computer-Human Interaction for documentation and modeling-tools. Specifically, automatic layout mechanisms for models are included under this area.

Changes to improve empirical quality of a model do not change the statements that is included in the model, thus we have no set-theorethic definition of this quality goal.

For informal textual models, several means for readability have been devised, such as the readability index. Over 50 procedures have been devised that claim to be able to compute how difficult a text is to read. As an example, the 'Fog Index' is arrived at in four steps:

1. Select several 100-word samples from a text.
2. Calculate the average sentence length by dividing the number of words by the number of complete sentences.
3. Obtain the percentage of long words in the entire sample: count the number of words containing three or more syllables and divide this total by the number of 100-words samples
4. Add 2 and 3 and multiply with 0.4. [1]

Several such formulae have been proposed, of varying level of complexity. Most assume that difficulty can be measured simply in terms of the length of the words and/or sentences. Factors such as the complexity of sentence construction and the nature of word meaning is often found to be much more important, but these the procedures usually ignore. Readability formulae have thus attracted a great deal of criticism, but in the absence of more sophisticated measures, they continue to attract widespread use, as a reasonably convenient way of predicting (though not explaining) reading difficulty.

For computer-output specifically, many of the principles and tools used for improved human computer interfaces are relevant at this level. Other general guidelines regards not mixing different fonts, colors etc. [338] in a paragraph that is on the same level within the overall text. Another set of techniques which can be useful at the empirical level, is those devised in so-called information theory [336].

[1] The product is the (American) grade-level for which the text is appropriate, in terms of difficulty.

For graphical models in particular, layout modification is a kind of meaning-preserving transformation which can improve the comprehensibility of models. A layout modification a spatially different arrangement of a diagrammatical representation of a model.

Lists of guidelines for graph aesthetics are presented in [23, 363], summarized in Table 3.2, and this could be a possible starting point for automatic layout modification.

Table 3.2. A taxonomy of graph aesthetics (From [363])

Aspect	Explanation
ANGLE	Angles between edges should not be too small.
AREA	Minimize the area occupied by the drawing.
BALANCE	Balance the diagram with respect to the axis.
BENDS	Minimize the number of bends along the edges.
CONVEX	Maximize the number of faces drawn as convex polygons.
CROSSING	Minimize the number of crossings between edges.
DEGREE	Place nodes with high degree in the center of the drawing.
DIM	Minimize differences among nodes' dimensions.
LENGTH	Minimize the global length of edges.
MAXCON	Minimize length of the longest edge.
SYMMETRY	Have symmetry of sons in hierarchies.
UNIDEN	Have uniform density of nodes in the drawing.
VERT	Have verticality of hierarchical structures.

Another use can be to produce metrics, e.g. the number of crossing lines divided by the number of links in total in a figure, or compared with the minimum possible number of crossing as long as one do not duplicate symbols. Similar metrics can be devised for the other aesthetics, and be used during modeling to assess the potential for improving aesthetics. Based on such metrics one could assess that the quality of Fig. 3.4 is less than that of Fig. 3.3 on this account although it contains the same statements.

On the other hand, we should remember that aesthetics is a subjective issue, thus familiarity with a diagram is often just as important for comprehension. As noted by [303] one of the main advantages of diagrammatic modeling languages appears to be the use of so-called secondary notation, i.e. the use of layout and perceptual cues to improve comprehension of the model. Thus, one often need to constrain automatic layout modifications. Although this more correctly should be placed as a means for pragmatic quality we include it here since it is used in techniques for automatic graph layout. A list of constraints used in connection to this is given in Table 3.3. Obviously, it should be easy to retain the aesthetically pleasing diagram when we have to update the model at a later point in time. This includes the possibility of selecting and moving a group of nodes as one, the moving of complete subtrees as one in a hierarchical model, re-routing connections when changing

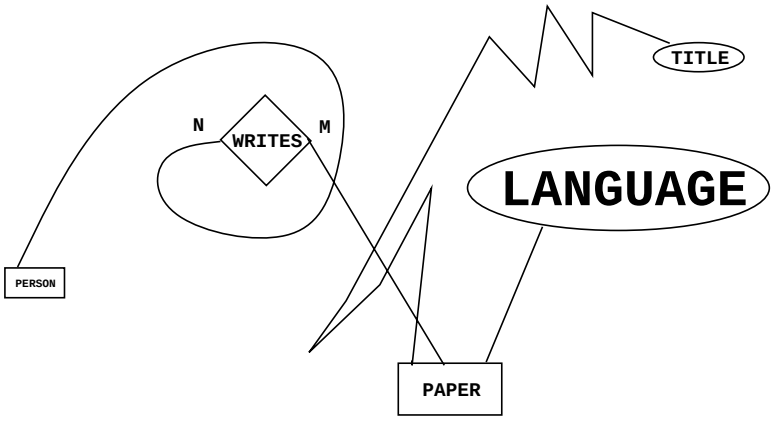


Fig. 3.4. Example on poor aesthetics

the relative position of two interconnected nodes, and functionality such as snap-to-grid.

Table 3.3. A taxonomy of constraints for graph layout (From [363])

Aspect	Explanation
CENTER	Place a set of given nodes in the center of the drawing.
DIMENSION	Assign the dimensions of symbols.
EXTERNAL	Place specified nodes on the external boundary of the drawing.
NEIGH	Place a group of nodes close.
SHAPE	Draw a subgraph with a predefined shape.
STREAM	Place a sequences of nodes along a straight line.

3.5 Syntactic Quality

Syntactic quality is the correspondence between the model $\mathcal{M}$ and the language extension $\mathcal{L}$ of the language in which the model is written. When looking at textual models, this include both lexicon correctness, syntax correctness and structural quality [108]. There is only one syntactic goal, **syntactical correctness**, meaning that all statements in the model are according to the syntax and vocabulary of the language i.e.

$$\mathcal{M}_E \setminus \mathcal{L} = \emptyset \tag{3.2}$$

Syntax errors are of two kinds:

– **Syntactic invalidity**, in which words or graphemes not part of the language are used. An example is given in Fig. 3.5, where an actor-symbol not being part of the language is introduced.

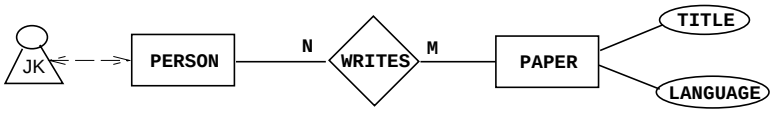


Fig. 3.5. Example of syntactic invalidity

– **Syntactic incompleteness**, in which the model lacks constructs or information to obey the language's grammar. An example is given in Fig. 3.6, where only one of the entity-classes that take part in the relationship is indicated.

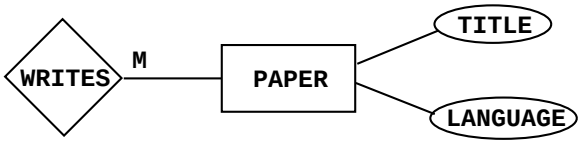


Fig. 3.6. Example of syntactic incompleteness

The degree of syntactic quality can be measured as one minus the rate of erroneous statements, i.e.

$$syntactic\ quality = 1 - \frac{\#(\mathcal{M}_E \setminus \mathcal{L}) + \mathcal{M}_{missing}}{\#\mathcal{M}_E} \tag{3.3}$$

where $\mathcal{M}_{missing}$ is the number of statements that would be necessary to make the model syntactically complete.

3.6 Semantic Quality

Semantic quality is the correspondence between the model and the modeling domain [239].

The framework contains two semantic goals; validity and completeness.

– **Validity** means that all statements made in the model are regarded as correct and relevant for the problem, i.e.

$$\mathcal{M} \setminus \mathcal{D} = \emptyset \tag{3.4}$$

A definition for the degree of validity could be

$$validity = 1 - \frac{\#(\mathcal{M}_E \setminus \mathcal{D})}{\#\mathcal{M}_E} \tag{3.5}$$

however, it can be questioned how useful such a metric might be, since it can never be measured due to the intractability of the domain. An example of invalidity is given in Fig. 3.7 where the attribute 'maximum speed' is

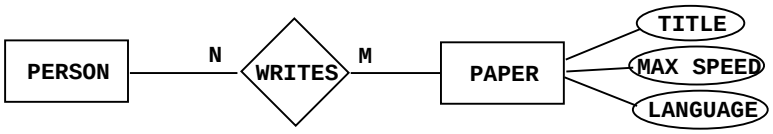


Fig. 3.7. Example of semantic invalidity

added to the entity 'paper', something we believe most persons would agree is invalid.

– **Completeness** means that the model contains all the statements which would be correct and relevant about the domain, i.e.

$$\mathcal{D} \setminus \mathcal{M} = \emptyset \tag{3.6}$$

A definition for the degree of completeness could similarly be

$$completeness = 1 - \frac{\#(\mathcal{D} \setminus \mathcal{M})}{\#\mathcal{D}} \tag{3.7}$$

This would only be interesting in limited domains, say e.g. that it is temporarily decided upon a model of a new CIS. Then one would like to see all the statements in the model also being part of the implemented CIS. On the other hand, $\mathcal{D}$ is not completely held in the previous model in this case, thus validity is also in here more relevant. An example of incompleteness can be the original Fig. 3.3, missing 'name' as an attribute of 'person', something we believe most persons would regard as important to represent in a conference system.

For anything but extremely simple and highly inter-subjectively agreed domains, total validity and completeness cannot be achieved. Hence, for the semantic goals to be realistic, they have to be somewhat relaxed, by introducing the idea of *feasibility*. Attempts at reaching a state of total validity and completeness will lead to unlimited spending of time and money on the modeling activity. The time to terminate a modeling activity is thus not when the model is "perfect" (which will never happen), but when it has reached a state where further modeling is regarded to be less beneficial than applying the model in its current state. Accordingly, a relaxed kind of validity and completeness can be defined, which we term feasible validity and completeness.

– **Feasible validity:** $\mathcal{M} \setminus \mathcal{D} = \mathcal{R} \neq \emptyset$, but there is no statement $r \in \mathcal{R}$ such that the benefit of performing a syntactically valid delete of r from $\mathcal{M}$ exceeds the drawback eliminating the invalidity r.
– **Feasible completeness:** $\mathcal{D} \setminus \mathcal{M} = \mathcal{S} \neq \emptyset$, but there is no statement $s \in \mathcal{S}$ such that the benefit of inserting s in $\mathcal{M}$ in a syntactically complete way exceeds the drawback of adding the statement s.

Feasibility thus introduces a trade-off between the *benefits* and *drawbacks* for achieving a given model quality. We have used the term "drawback" here instead of the more usual "cost" to indicate that the discussion is not necessarily restricted to purely economical issues. Judging completeness with respect to some inter-subjectively agreed standard as suggested by Pohl [306] is one approach to feasibility. By making the standard a part of the language, one can in addition transfer a semantic problem into a syntactic one.

3.7 Perceived Semantic Quality

Perceived semantic quality is the correspondence between the actor interpretation of a model and his or hers current knowledge of the domain. Perceived validity and completeness can be expressed as indicated below:

- *Perceived validity* of the model externalization: $\mathcal{I}_i \setminus \mathcal{K}_i = \emptyset$.
- *Perceived completeness* of the model externalization: $\mathcal{K}_i \setminus \mathcal{I}_i = \emptyset$.

Metrics for the degree of perceived validity and completeness can be defined by means of cardinalities the same ways as for syntactic quality.

$$\textit{perceived validity} = 1 - \frac{\#(\mathcal{I}_i \setminus \mathcal{K}_i)}{\#(\mathcal{I}_i)} \tag{3.8}$$

I.e. the number of invalid statements interpreted, divided by the total number of statements interpreted by the actor A_i. An example of a model with a perceived invalid statement is the example in Fig. 3.3, where I, in the role of an end-user of a conference system, would not regard the 'language' attribute to be relevant for 'paper'.

$$\textit{perceived completeness} = 1 - \frac{\#(\mathcal{K}_i \setminus \mathcal{I}_i)}{\#(\mathcal{K}_i)} \tag{3.9}$$

I.e. the number of relevant statements known, but not seen in the model, divided by the total number of relevant knowledge statements known by the actor A_i. Also on these measures, a discussion of feasibility is useful. As on semantic quality, I miss among other things the name of person in the model in Fig. 3.3.

The perceived semantic quality of the model can change in many ways:

- A statement is added to $\mathcal{M}^i$ which is understood to be in accordance with the knowledge of actor A_i, thus increasing perceived completeness.
- A statement is added to $\mathcal{M}^i$ which is understood to not be in accordance with the knowledge of actor A_i, thus decreasing perceived validity.
- A statement is removed from $\mathcal{M}^i$ that earlier was understood not to be in accordance with the knowledge of actor A_i, thus increasing perceived validity.

- A statement is removed from $\mathcal{M}^i$ that earlier was understood to be in accordance with the knowledge of actor A_i, thus decreasing perceived completeness.
- $\mathcal{K}_i$ changes, which can both increase and decrease perceived validity and completeness of the model. One way $\mathcal{K}_i$ can change, is through the internalization of another model made on the basis of the knowledge of another actor. Internalization will be discussed further after discussing social quality below.
- The actor's knowledge of the modeling language changes, potentially changing $\mathcal{I}_i$ which can both increase and decrease the perceived validity and completeness of the model.

3.8 Pragmatic Quality

Pragmatic quality is the correspondence between the model and the audience's interpretation of it. The framework contains one pragmatic goal, namely **comprehension**. Not even the most brilliant solution to a problem would be of any use if nobody was able to understand it. Moreover, it is not only important that the model has been understood, but also *who* has understood it.

Individual comprehension is defined as the goal that the individual actor A_i understands the part of the model relevant to that actor, i.e. $\mathcal{I}_i = \mathcal{M}^i$.

The corresponding error class is *incomprehension*, meaning that the above formula does not hold.

For a large model, it is unrealistic to assume that each audience member will be able to comprehend the consequences of all the statements which are relevant to them. Thus, comprehension as defined above is an ideal goal, just like validity and completeness, and can often not be achieved. Again it will be useful to introduce the notion of feasibility:

Feasible comprehension means that although the model may not have been correctly understood by all audience members, i.e.

$$(\exists i)(\mathcal{I}_i \setminus \mathcal{M}^i) \cup \ (\mathcal{M}^i \setminus \mathcal{I}_i) = \mathcal{S}_i \neq \emptyset, \tag{3.10}$$

there is no statement $s \in \mathcal{S}_i$ such that the benefit of rooting out the misunderstanding corresponding to s exceeds the drawback of taking that effort.

It is important to notice that the pragmatic goal is stated as *comprehension*, i.e. that the model has been understood, not as *comprehensibility*, i.e. the model's ability to be understood. There are several reasons for doing so. First, the ultimate goal is that the model is understood, not that it is understandable. Moreover, it is hard to speak about the comprehensibility of a model as such, since this is very dependent on the process by which it is developed, the way the participants communicates with each other and

various kinds of tool support.This is also the main reason for differentiating between empirical and pragmatic quality.

3.9 Social Quality

The goal defined for social quality is *agreement*. Six kinds of agreement can be identified, according to the following two orthogonal dimensions:

- Agreement in knowledge vs. agreement in model interpretation. In the case where two models are made based on the view of two different actors, we can also talk about agreement in model.
- Relative agreement vs. absolute agreement.

Relative agreement means that the various projections or models are consistent – hence, there may be many statements in the projection of one actor that are not present in that of another, as long as they do not contradict each other. Absolute agreement, on the other hand, means that all projections are the same.

Agreement in model interpretation will usually be a more limited demand than agreement in knowledge, since the former one means that the actors agree about what is stated in the model, whereas there may still be much they disagree about which is not stated in the model so far, even if it might be regarded as relevant by one or both participants. On the other hand, agreement of models will be easier to check in practice especially if the languages have formal syntax or semantics, although this is limited to the situation as described above.

Hence, we can define

- Relative agreement in interpretation: all I_i are consistent,
- Absolute agreement in interpretation: all I_i are equal,
- Relative agreement in knowledge: all K_i are consistent,
- Absolute agreement in knowledge: all K_i are equal,
- Relative agreement in model: all M_i are consistent,
- Absolute agreement in model: all M_i are equal,

Metrics can be defined for the degree of agreement based on the number of inconsistent statements divided by the total number of statements perceived, or by the number of non-corresponding statements divided by the total number of statements perceived.

Since different participants will have their expertise in different fields, relative agreement is regarded to be more useful than absolute agreement. On the other hand, the different actors must have the *possibility* to agree and disagree on something, i.e. the parts of the model which are relevant to them should overlap to some extent.

It is not given that all participants will come to agreement. Few decisions are taken in society under complete agreement, and those that are are not

necessarily good, due to group-think and other detrimental factors. To answer this we introduce *feasible agreement*:

Feasible agreement is achieved if feasible perceived semantic quality and feasible comprehension are achieved and inconsistencies are resolved by choosing one of the alternatives when the benefits of doing this are greater than the drawbacks of working out an agreement.

3.10 Knowledge Quality

From a standpoint of social constructivity, it is difficult to talk about the quality of explicit knowledge. On the other hand, within certain areas, for instance mathematics, what is generally regarded as "true" is comparatively stable, and it is inter-subjectively agreed that certain persons have more valid knowledge of an area than others. It is important to keep in mind that their knowledge is only partial. The "quality" of the participant knowledge can thus be expressed by the relationships between the audience knowledge and the domain. The "perfect" situation would be if the audience knew everything about the domain at a given time.

$$Knowledge\ completeness : \mathcal{D} \setminus \mathcal{K} = \emptyset \tag{3.11}$$

and that they had no "incorrect" superstitions about the domain, i.e.,

$$Knowledge\ validity : \mathcal{K} \setminus \mathcal{D} = \emptyset \tag{3.12}$$

To get a *good enough* knowledge about the domain, careful *participant selection* based on *stakeholder identification* is necessary (if you have a problem and can choose the participants), or alternatively, careful *problem selection* (if the participants are given, but not the problem to be solved). In the case that both participants and problem are more or less given, and not fitting too well, some development in terms of training of the participants or modification of the problem may be necessary. Just as for the other aspects of quality, it will be possible to talk about *feasible knowledge quality*, meaning that the knowledge of the audience could still be improved, but the benefit of improving it through additional education or the hiring of additional experts or including additional stakeholders will be less than the drawbacks of mistakes made due to imperfect knowledge. In the view of social construction every stakeholder might have something unique to contribute to the process of conceptual modeling. On the other hand, it is obviously not feasible to include, say, 500 end-users and 200000 indirect users (customers) of the organization in constructing the requirements of a new application system.

3.11 Quality of Conceptual Modeling Languages

The discussion is based on the overview originally made by Sindre [344], and extended by Seltveit [331]. Their results are rearranged to fit with the

categories in the framework for model quality. The given criteria are to some extent a matter of belief and taste, but is generally based on results from linguistics and psychology [6, 247, 352, 410].

In the original framework, distinctions were made along two dimensions. First, it was distinguished between two main kinds of criteria:

- Criteria for the underlying (conceptual) basis of the language, i.e. the constructs of the language.
- Criteria for the external representation of the language, i.e. how these constructs are represented visually.

For each of these two parts, the following four main groups of criteria were identified:

- **Perceptibility**: How easy is it for persons to comprehend the language? This is related to the current and potential knowledge of the participants and their interpretation of models in the language.
- **Expressive power**: What is it possible to express in the language? This is related to the domain.
- **Expressive economy**: How effectively can things be expressed in the language? This is also related to participant interpretation.
- **Method/tool potential**: How easily does the language lend itself to proper method and tool support? This is related to the capabilities of the technical actors in the audience.

In addition, Seltveit introduced reducibility as a separate category, meaning what features is provided by the language to deal with large and complex models.

We have regrouped the factors according to the framework for model quality as follows:

- Domain appropriateness. Relates the language to the domain and vice versa.
- Participant language knowledge appropriateness. Relates the participant knowledge to the language.
- Knowledge externalizability appropriateness. Relates the language to the participant knowledge.
- Comprehensibility appropriateness. Relates the language to the social audience interpretation.
- Technical actor interpretation appropriateness. Relates the language to the technical audience interpretations.

In Fig. 3.8 it is indicated how these relationships are related to the sets in the quality framework.
Since participant interpretation is done on the basis of the current participant knowledge, factors 2 and 3 will be closely intertwined.

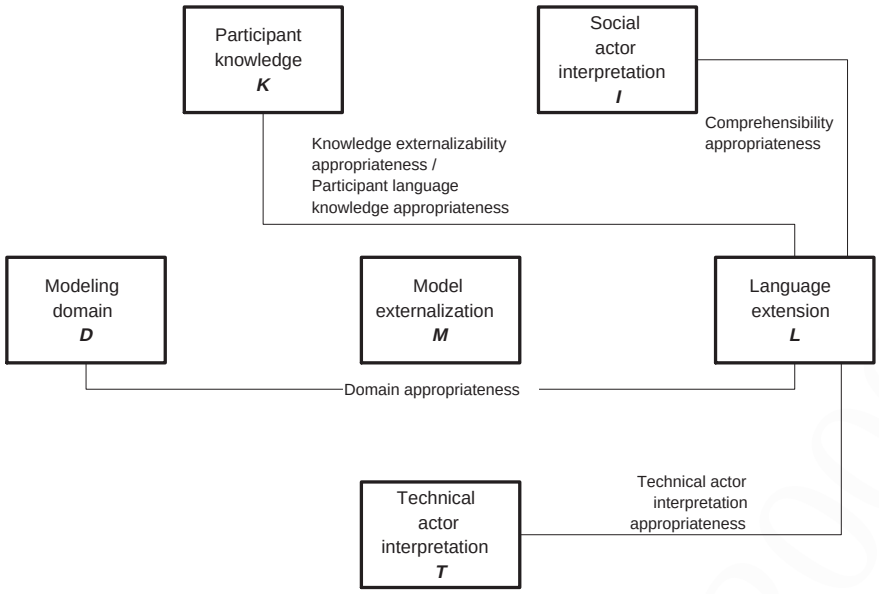


Fig. 3.8. Coverage of this section

We will continue to distinguish between the underlying basis of a language and its external representation since it will result in a clearer discussion. Different criteria in the different factors will often be contradictory, i.e. one should expect to find some deficiencies for most conceptual modeling languages based on goals for language quality. On the other hand, this can be addressed by how the language is used within a methodology, including the use of specific modeling techniques.

3.11.1 Domain Appropriateness

Domain appropriateness can be describes as follows:

$$\mathcal{D} \setminus \mathcal{L} = \emptyset \tag{3.13}$$

i.e. there are no statements in the domain that cannot be expressed in the language. Obviously this means that different languages are more or less suitable for different problem situations. In Chap. 2.2 the existing perspectives used and deemed necessary to support a wide range of modeling situations was discussed. This general notion is parallel to what is termed expressiveness in [280]. Domain appropriateness on more limited areas is discussed in detail by many. An early comparison of data modeling languages is given in [301]. Davis gives an overview of requirements for languages for modeling external system behavior in [79]. Embley et al. have a similar overview of aspects of object-oriented modeling languages for analysis in [106]. Iivari compares

different object-oriented languages according to how good they are for modeling structural, functional, and behavioral aspect in [178]. In [54], domain appropriateness of languages for modeling workflow systems is discussed.

- Underlying basis: Ideally, the basis must be powerful enough to express anything in the domain. On the other hand you should not be able to express things that are not in the domain, if this makes you focus on the wrong aspects. A typical example on this is the difference between analysis and design models. A language which is very good for modeling a design, might not be too good for analysis, since it will need to include computerized information system specific constructs that if used in analysis can result in a product rather than a problem-oriented model [106].
  There is an infinite number of statements that we might need to make, and these have to be dealt with through a rather small number of phenomena classes due to the comprehensibility appropriateness, as will be discussed below. This means that
  - The phenomena must be *general* rather than specialized. This is parallel to what is termed genericity in [280].
  - The phenomena must be *composable*, which means that we can group related statements in a natural way. When only domain appropriateness is concerned, it is an advantage if all thinkable combinations are allowed, each yielding a separate meaning.
  - The language must be *flexible in precision:*
    - To express precise knowledge one needs precise constructs. This means that the language must be *formal* and *unambiguous*.
    - At the same time, one need vague constructs for modeling vague knowledge. To fulfill both requirements, the vagueness must also be formalized (i.e. even the vague constructs must have a definite interpretation – the constructs are called vague because their interpretation is wide compared to the more definite constructs).
- External representation: The only requirement to the external representation is that it does not destroy the underlying basis. Thus,
  - Every possible statement in the language should have a unique representation in the basis (otherwise, the precision of the language will be destroyed).
  - Every possible statement in the underlying basis should have at least one external representation (otherwise, the generality might be destroyed).
  - Just like the phenomena, the symbols of the language must be *composable*.

  As indicated, only the mapping from symbols to phenomena needs to be unique – it is all right to have several alternative external representations for the same statement.

3.11.2 Participant Knowledge Appropriateness

The overall goal here is that all the statements in the language models of the languages used by the different participants are part of the explicit knowledge of this participant.

– Underlying basis: This should correspond as much as possible to the way individuals perceive reality. This will differ from person to person and between persons in different groups according to their previous experience [136], and thus will initially be directly dependant on the given participants in a modeling effort. When it comes to existing use of modeling languages, Senn [332] reports that the level of awareness of structured methods (i.e. using data and process modeling languages) is high among CIS-professionals - as many as 90 percent of all analysts are familiar with these methods, according to some estimates. Approximately half of the organizations in the United States have used these methods.
On the other hand the knowledge of the participants is not static, i.e. it is possible to educate persons in the use of a specific language. In that case should one base the language on experience with languages for conceptual modeling, and languages that have been used successfully earlier in similar tasks. In this connection, it is interesting to look on experiments trying to find which languages or perspectives persons find most easy to learn. Few empirical studies of this kind have been performed. Vessey and Conger [381] reports in empirical investigations among novice analysts that they seemed to have much greater difficulty applying an object methodology, than a data or process methodology. Process modeling was found easier to apply than data modeling. For experienced developers the most difficult legacy to overcome before being able to use object-orientation efficiently seems to be the investment in persons whose experience and expertise are in other ways of doing things. According to Kozaczynski, to become accepted, the object-oriented way of thinking must become the natural way of thinking. Now it presents a steep learning curve [205]. In Tempora [367], the experience was that whereas participants had small problems in learning to use both the process modeling language and the main parts of the data modeling language, the formal textual rule language was difficult for people to comprehend. On the other hand, the phenomena of rules is generally well-known: As stated by Twining [373], "One reason why the notion of 'rule ' is such an important one not only in law, but in fields as varied as linguistics, sociology, anthropology, education, psychology, and philosophy, is that there is hardly any aspect of human behavior that is not governed or at least guided by rules." When it comes to the communication perspective, some experience related to learning speech-acts theory as part of using tools such as the Coordinator [118] is presented in [48]. In many cases, the users found the linguistically motivated parts of the language difficult to understand and apply. In other cases, this was not regarded as a problem.

When it comes to actors and roles, we believe these phenomena classes to be easy to comprehend based on their widespread use in e.g. organizational diagrams.
Another important point in this connection is that it should be possible to express inconsistencies and dispute in the language since inconsistency between how people perceive reality is a fact of life which is useful to represent so that these can be revealed and discussed explicitly.

– External representation: The external representation of different phenomena should be *intuitive* in the sense that the symbol chosen for a particular phenomenon somehow reflects this better than another symbol would have done. Also this is partly dependent on the audience, even if general guidelines might be devised. For instance, it can be useful to represent areas in a model that is not complete using a specific notation such as the use of wiggly lines [132] or amoeba shapes [344].

3.11.3 Knowledge Externalizability Appropriateness

Knowledge externalizability appropriateness can be describes as follows:

$$\mathcal{K} \setminus \mathcal{L} = \emptyset \tag{3.14}$$

i.e. there are no statements in the explicit knowledge of the particpant that can not be expressed in the language. This focuses on how relevant knowledge may be articulated in the modeling language. This is clearly linked to concepts like "articulation work" and "situated action" and the ongoing debate in Computer Supported Cooperative Work (CSCW) whether actual work can be captured in a model, or whether such a model always will be a post-hoc rationalization [355]. This aspect of language quality will not be looked into in detail in this paper. Again this mean is primarily supporting the achievement of physical quality.

3.11.4 Comprehensibility Appropriateness

Similar to model interpretation, one can define language interpretation, thus the set of possible statements that can be made in the language that are understood by the audience member. Ideally

$$\mathcal{L} \setminus \mathcal{I} = \emptyset \tag{3.15}$$

i.e. all the possible statements of the language are understood by the particpants in the modeling effort using the language.

– Underlying basis:
  – The phenomena of the language should be easily distinguished from each other.

- The number of phenomena should be reasonable. If this has to be uncomfortably large, they should be organized hierarchically, making it possible to approach the conceptual framework at different levels of sophistication. This hierarchical organization should in itself be natural, cf. the participant knowledge above.
- The use of phenomena should be *uniform* throughout the whole set of statements possible to express within the language. Using the same construct for different phenomena or different constructs for the same function depending on the context will tend to make the language confusing.
- The language must be flexible in *the level of detail:*
  - Statements must be *easily extendible* with other statement providing more details.
  - At the same time, details must be *easily hidden*.

  This means that the language should include abstraction mechanisms.
- Separation of concerns: It is possible to divide the models made in the language in natural parts, to be able to support work division in the sense that the individual participants can concentrate on the areas they are interested in.

Whereas the domain appropriateness concerns what we are able to express, *expressive economy* concerns how briefly things can be expressed, i.e. how many constructs you need to make the statements you want to make. Introducing one construct for each possible statement would make every statement brief, but the number of constructs would be infinite. Since it is necessary to keep the number of construct at a reasonable level for people to be able to learn the language, a good expressive economy cannot be based on defining new constructs for everything. Instead

- The most *frequent* kinds of statements should be as brief as possible.
- The most *important* kinds of statements should be as brief as possible.

- External representation:
  - Symbol discrimination should be easy. This means that it should be easy to see the difference between the various symbols of the language.
  - The external language should be as consistent as possible, in the sense that symbol use should be uniform, i.e. a symbol should not represent one phenomenon in one context and a completely different one in another. Neither should one use different symbols for the same phenomenon in different contexts.
  - One should strive for symbolic simplicity – both concerning the primitive symbols of the language and the way they are supposed to be connected. If the symbols themselves are visually complex, models containing a lot of symbols will be even more complex, and thus difficult to comprehend.
  - The use of emphasis in the external language should be in accordance with the relative importance of the statements. Factors that have an important impact on visual emphasis are the following:

- Size (the big is more easily noticed than the small). On the other hand, nodes in a diagram should not have very different size.
- Solidity (e.g. **bold** letters vs. ordinary letters, full lines vs. dotted lines, thick lines vs. thin lines, filled boxes vs. non-filled boxes).
- Difference from the ordinary pattern (e.g. *slanted* letters or a rare symbol will attract attention in a model of ordinary ones).
- Foreground/background (if the background is white, things will be easier noticed the darker they are).
- Color (red attracts the eye more than other colors).
- Change (blinking or moving symbols attract attention).
- Pictures vs. text (pictures usually having a much higher perceptibility, information conveyed as such will be emphasized at the cost of information conveyed textually).
- Position (looking at a diagram, people tend to start at its middle).
- Degree (nodes able to connect to many others will attract attention compared to nodes making few connections).

Areas such as size and color might also be model-specific, in case the modeling tool enables resizing etc. Emphasis is a very powerful mechanism for facilitating comprehension of models, but it can easily be overdone.

– Composition of symbols can be made in a aesthetically pleasing way, such that one can enable the creation of models with few crossing edges and short edges. The possibility of model redundancy can also be important in this respect.
– Navigation: Does the external constructs allow for nice ways of filtering, i.e. making various selections concerning which statements to show and which to hide, and browsing, i.e. moving between related symbols in a model?
– Grouping of related statements: Does the language have constructs to support the grouping of related statements?

Graphical styles of representation have some important advantages over text and tables when it comes to enhancing participant interpretation based on the possible use of secondary notation, i.e. the use of layout and perceptual cues such as adjacency, clustering, alignment, and white space [303].

At the external level, expressive economy is concerned with how many symbols one need to use to express the statements of the model. As the requirements of the previous item suggest, it will usually be the case that the things easily expressed conceptually will also be easily expressed externally, and the things which are complicated in the underlying basis will also have to be complicated externally.

However, the basis and the external representation of a language should not necessarily be the same. A good external representation should always have an expressive economy better than that of the basis. This is because

the external representation has many possibilities that the underlying basis does not have:
- Omission of symbols that are understood in the context.
- Special symbols can be defined for constructs which are frequent (or important).
- Multiple mentioning of the same phenomena is unavoidable at the basis level. At the external level, such multiple mentioning can often be avoided.

Of course, there are some pitfalls to avoid.
- Blank symbols, i.e. symbols that do not contain any information for anyone.
- External redundancy, i.e. showing the same phenomena in several different ways in the same external representation.

Diagrams have a significantly larger potential for expressive economy than tables or text. On the other hand, it is impossible to convey everything diagrammatically. Thus, the best thing to do for expressive economy is to try to express the frequent and most important statements diagrammatically and the less frequent textually.

Another aspect in connection to comprehensibility appropriateness is that of liberality [280], being defined as the degree of freedom one has when modeling the same domain. On the one hand, high liberality is useful, since it make it possible to rephrase a model in many different ways. On the other hand, liberality might cause confusion since it might be more difficult to see the similarities of two models. The use of the view integration technique as presented in Chap. 7 is one way of addressing this problem.

3.11.5 Technical Actor Interpretation Apropriateness

- Underlying basis: For the technical actors, it is especially important that the language lend itself to automatic reasoning. This requires formality (i.e. both formal syntax and semantics being operational and/or logical), but formality is not necessarily enough, since the reasoning must also be fairly efficient to be of practical use. This is covered by executability.
  On the other hand, a model expressed in natural language can also be useful from this point of view as techniques for natural language understanding [127] are improved. As an example, Moulin and Rousseau describes a system which constructs formal rules automatically from regulation texts [265]. In ALECSI [57, 316], a natural language description is automatically translated into a simple semantic net using a set of standardized sentence patterns. VDM models have been developed from structured analysis models [125], and TELL (a language based on temporal logic) models from larger bodies of NL text [321].
- Information hiding constructs: Encapsulating parts of the model limits its access from other components, i.e. to create independent parts. This is

a useful property when the models are used to generate the application system, and thus simplifies testing. It is also useful when one want to focus on only some part of the model.

Techniques taking advantage of the technical appropriateness of a language will be discussed in the next three chapters.

3.12 Quality of a Software Requirements Specifications (SRS)

As mentioned in the introduction of the chapter, Davis et al. summarizes the work on quality attributes for a software requirements specifications (SRS), giving the most comprehensive list of such properties in literature in [81]. The paper also includes proposals for metrics and weights for the different properties. The following properties are discussed by Davis:

- Unambiguous
- Complete
- Correct
- Understandable
- Verifiable
- Internally consistent
- Externally consistent
- Achievable
- Concise
- Design independent
- Traceable
- Modifiable
- Electronically stored
- Interpretable
- Prototypable
- Annotated by relative importance
- Annotated by relative stability
- Annotated by version
- Not redundant
- At right level of detail
- Precise
- Reusable
- Traced
- Organized
- Cross-referenced.

An SRS can be looked upon as being either a model of the perceived future IS, or the perceived future CIS without locking it to one specific implementation. In either case, the domain also involves statements about the

resource constraints for this particular development effort, together with already baselined documents and models created earlier in the development effort.

In an SRS, one usually use a mix of conceptual models and natural language text, thus it is necessary to include quality means for both kinds of models in combination. Thus, in addition to our own work and the work of Davis, we apply work done on quality for textual models /citeFabbrini:98. One important aspect when discussing requirements models, is that they are meant to be understood by persons with very varying background (compared to e.g. design models which are normally only used by persons with detailed software development knowledge). We discuss means within each quality level in detail, starting with those areas that are specifically mentioned by Davis. We have highlighted these properties in boldface when positioning them within the quality framework below. There will be some overlap here with the genral discussion earlier in the chapter, and this is done for completeness of the coverage in this chapter.

3.12.1 Physical quality of an SRS

The only property in this area mentioned by Davis is that the SRS should be **electronically stored**. This is covered by the persistence mean for addressing the physical quality aspect of internalizeability. Important means for achieving this are using modeling languages that are appropriate for the domain and participant knowledge as discussed above under language quality. An important aspect in relation to a requirement specification is that the languages used should not put unneccesary constraints on the technical solution. Areas in the discussion of Davis that potentially influence this discussion are: **Design independent, Traceable, Annotated by relative importance, Annotated by relative stability, Annotated by version**, and **Precise**. Since these areas are not always covered and mandated as an integral part of the language, we will return to these areas in more detail as part of semantic quality below. One important activity in this area is the adaptation of the meta-model of the language used to suit the domain. This involves both adding concepts, but also removing concepts (temporarily) from the language if they are not relevant for the modeling of the particular domain. Internalizeability on the physical level has two primary means, persistence and availability: Many of the general activities in connection with physical quality are based on traditional database-functionality using a repository-solution for the internal representation of the model. In addition, it is regarded necessary for advanced tools for requirement specification to include functionality such as version control and configuration management, that are not normally found in conventional DBMSs.

3.12.2 Empirical quality of an SRS

Davis covers this area with the property **understandable**: An SRS is understandable if all classes of SRS readers can easily comprehend the meaning of all requirements with a minimum of explanation. An important factor in achieving this is language quality aspects of the modeling languages that have been used such as comprehensibility appropriateness. The property **Concise** (An SRS is concise if it is as short as possible without affecting any other quality of the SRS) is a mean on this level, often being related to the expressive economy of the language being used. It can also be linked to overall size limitation of the SRS. To further make Davis' goal more concrete, the same guidelines apply as for diagrammatical or textual models in general when it comes to empirical quality. For informal textual models, a range of means for readability has been devised, such as the readability index. For computer-output specifically, many of the principles and tools used for improved human-computer interface are relevant at this level. Other general guidelines regard not mixing different fonts, colors etc. in a paragraph being on the same level within the overall text. Another set of techniques which is often useful here, are those devised within information theory. For graphical models in particular, layout modification is found to improve the comprehensibility of models.

3.12.3 Syntactic quality of an SRS

This area is not really addressed by Davis, although some of the aspects on the semantic level can easily be reduced to syntactic issues by mandating certain aspects to be part of the language (e.g. priority, version, and stability information). Since Davis does not mention this explicitly as a mean, we will treat these properties as part of semantic quality. The main mean on this level is the use of a language with formal syntax (also mentioned by Davis), in which case it can be possible to provide different types of tool support to achieve syntactic quality. To assure the syntactic quality of the model, syntax checks should be provided as an integral part of the modeling support. The checks may be carried out along two main directions.

1. Error prevention, adapting the principle of syntax-directed editors, making it impossible to make certain types of syntactic errors.
2. Error detection, with explicit checking being enforced by the user and the check both detects and report existing errors, or also suggest how to correct the error. Errors due to syntactical incompleteness usually have to be checked in this fashion.

3.12.4 Semantic quality of an SRS

Most of the properties discussed by Davis concerns semantic quality. It is important to notice that these quality properties have been suggested under

the assumption of an objectivistic world-view. When comparing them with validity and completeness as we have defined them, we thus do this under the presumption that the modeling domain is inter-subjectively agreed among the members of the audience.

First, when looking upon semantic quality relative to the primary domain, we have the property **Complete**. An SRS is complete if

1. everything that the software is supposed to do is included in the SRS
2. Responses of the software to all realizable classes of input data in all recognizable classes of situations are included.
3. All pages numbered, all figures and tables numbered, named, and referenced; all terms defined; all units of measure provided; and all referenced material present.
4. No sections marked To be determined.

The first point is the same as our measure of completeness, whereas achieving the second is supported through the modeling activity of using the driving question technique. Item 3 and 4 on the other hand is a kind of incompleteness that potentially is easier to deal with. This can be done either by manually checking for such situation after including them as part of a standard document-structure to be followed for the SRS, or by including such aspects as part of the syntax of the modeling language. In this way, one is able to reduce an apparent semantic problem into one of checking for syntactic completeness and validity.

Correct: An SRS is correct if and only if every requirement represent something required of the system to be built. This is the same as validity.

The property **internally consistent** (An SRS is internally consistent if and only if no subset of individual requirements stated therein conflict) is subsumed by the combination of validity and completeness since an inconsistency must be caused by at least one invalid statement or the lack of a statement that are to sort out the inconsistency. To illustrate, consider the case in which you must model an organization' s business rules for implementing these in an information system. Suppose the system must account for the rule If a company has been our customer for more than 10 years, the customer status should be high priority and the rule If a customer has been late with payments more than three times, the customer status should be low priority. But what if a company who has been the customer for 12 years is late with payments more than three times? You could decide to change the first rule to rate the customer as medium priority or the second to more than four times. Either of these two actions would mean that the original rules were invalid. On the other hand, you might decide that both rules are valid, and add another rule if there are contradictory rules about customer status, the sales manager should resolve the issue, which requires the system simply to notify someone. Adding a rule to resolve a contradiction would mean that the original model was incomplete because it had no such rule. Davis suggests using languages with formal syntax and semantics to address inconsistency,

a mean for semantic quality also proposed by us. Note that semantic consistency checking might only detect inconsistencies, it will be up to human judgment if the consistency is because of invalidity or incompleteness of the model.

Another property being either a matter of incompleteness or invalidity relative to the primary domain is **precise** (An SRS is precise if and only if (a) numeric quantities are used whenever possible and (b) the appropriate levels of precision are used for all numeric quantities). The first aspect is covered by completeness. If the granularity of precision is too high, this can also be regarded as incompleteness, whereas if it is too low, there is a case of invalidity.

Properties related to the pre-existing context are:

- **Traced** (An SRS is traced if and only if the origin of each of its requirements is clear) is subsumed by completeness since such links to other models and/or sources of the requirements should be captured in the model if they are deemed relevant.
- **Externally consistent** (An SRS is externally consistent if and only if no requirement stated therein conflict with any already baselined project documentation). Statements within such documentation will be part of the pre-existing context; thus, the same can be said about external consistency as was said about internal consistency above.

The following properties are related to completeness and validity relative to the purpose context.

- **Annotated by Relative Importance**: An SRS is annotated by relative importance if a reader can easily determine which requirements are of most importance, which are next most important etc. Since this is needed to allocate resources sensibly, and determine priorities when budgets are inadequate, this is part of completeness.
- **Annotated by Relative Stability**: An SRS is annotated by relative stability if a reader can easily determine which requirements are most likely to change, which are next most likely etc. Since this is needed for designers to know where to build in flexibility, an SRS that is not annotated in this way is incomplete.
- **Annotated by Version**: An SRS is annotated by version if a reader can easily determine which requirements will be satisfied in which version of the product. When relevant, the lack of this information is also an example of incompleteness.

On the above three aspects, if it is decided that the language for modeling being used should contain such information (e.g. priority information in QFD [25] or deontic operators in a rule language [213]), the lack of this can rather be looked upon as an example of syntactic incompleteness than semantic incompleteness.

- **Traceable**: An SRS is traceable if and only if it is written in a manner that facilitates the referencing of each individual statement. This indicates requirements to the language to be used for modeling, thus if the decided language include these kind of aspect, a requirement specification missing them would be syntactically incomplete. If these aspects are not formally included in the language, one needs to treat them as problems of semantic completeness.
- **Verifiable**: An SRS is verifiable if there exist finite, cost effective techniques that can be used to verify that every requirement stated therein is satisfied by the system to be built. This is partly related to completeness, especially when the requirement is difficult to verify because of ambiguity (see also below). Problems with verifiability because of lack of precision are discussed under the property 'precise'. When verifiability is problematic because of undecidability, this should be explicitly stated if it is relevant.
- **Achievable**: An SRS is achievable if and only if there could exist at least one system design and implementation that correctly implements all the requirements stated in the SRS. Since it is part of the purpose of an SRS that it should be transformed (usually manually) into a computerized information system, an SRS that is not achievable is invalid. This specific kind of invalidity calls for specific means such as proof of concept through prototyping.
- **Design-independent**: An SRS is design-independent if and only if there exist more than one system design and implementation that correctly implements all requirements stated in the SRS. This is covered by validity, since if the SRS was not design-independent, it would be over-constrained, and these extra constraints can be looked upon as invalid statements in an SRS.
- **At right level of detail**: Requirements can be stated at many levels of abstractions. The right level of detail is a function of how the SRS is being used. Generally, the SRS should be specific enough so that any system built that satisfies the requirements in the SRS satisfies all user needs, and abstract enough so that all systems that satisfy all user needs also satisfy all requirements. This indicate that the requirements specification need to be complete, and not over-constrained, i.e. valid, as discussed earlier, thus no new aspects are really included by this property.

Unambiguous: An SRS is unambiguous if and only if every requirement stated therein has only one possible interpretation. This is subsumed by validity and completeness: If the model is consistent and valid, nothing is wrong with having ambiguity, expect that you should state explicitly that all alternative interpretations are intended. Without this explicit statement, there is incompleteness related to the purpose context. Note that the property 'design independent' in fact necessitate a certain degree of ambiguity. Davis suggest the use of formal languages to address ambiguity, similarly to us indicating that using modeling languages with a formal semantics is a mean

for achieving semantic quality. Some additional semantic means mentioned as properties by Davis are:

Modifiable: An SRS is modifiable if its structure and style are such that any changes can be made easily, completely and consistently. To improve the semantic quality of a model, one needs to change the model. This includes both the cases where the model is found invalid or incomplete in relation to a stable domain, or when it is the domain that changes, e.g. when the requirements to the system change. In connection to this, we have the property **Not redundant** (An SRS is redundant if the same requirement is stated more than once) Unlike the other properties, redundancy is not necessarily bad. Redundancy can in fact increase pragmatic quality (see below). The main problem of redundancy hits when SRS is changed. Thus, avoiding uncontrolled redundancy is a (secondary) mean to achieve modifiability.

3.12.5 Pragmatic quality of an SRS

Some other properties mentioned by Davis on this area are:

Executable/Interpretable/Prototypable: An SRS is executable, interpretable, or prototypable if and only if there exists a software tool capable of inputting the SRS and providing a dynamic behavioral model. To perform the indicated activities, one obviously need tool support for models developed in languages having an operational semantics, although the existence of the tool support is not a quality feature of the model itself.

Organized: An SRS is organized if and only if its contents are arranged so that readers can easily locate information and logical relationships among adjacent sections are apparent. One way is to follow any of many SRS standards, e.g. group by class of user, common stimulus, common response, feature, or object.

Cross-referenced: An SRS is cross referenced if and only if cross-references are used in the SRS to relate sections containing requirements to other sections containing: Identical (i.e. redundant) requirements, more abstract or more detailed descriptions of the same requirements and requirements that depend on them or on which they depend. As discussed earlier, such links are needed to assure the comprehension of the overall model; thus having them can be classified as a pragmatic mean. They are also related to modifiability. Using e.g. hyperlinks and having advanced browsing capabilities are useful tool support in this area. There are a number of other activities that can support pragmatic quality, such as audience training, inspections and walkthroughs, model rephrasing, model filtering, animation, explanation generation, and simulation.

3.12.6 Social quality of an SRS

Davis does not address this area specifically. Tool support in this respect is most easy to device on achieving agreement in models created based on the

internal reality of the participants that are to agree. One can also support the specific process of achieving feasible agreement. Based on this, main activities for achieving feasible agreement are model integration with specific emphasis on conflict resolution between the models to be integrated. Argumentation tools are also useful at this level.

3.12.7 Orthogonal Aspects

Finally, there is one of the properties suggested by Davis that can be looked upon across all the semiotic levels namely **Reusable**: An SRS is reusable if and only if its sentences, paragraphs, and sections can be easily adopted and adapted for use in subsequent SRS. This is dependent on many factors:

- The model needs to have good physical quality i.e. be physically represented in a persistent form that is available to those who potentially will want to reuse it.
- For reuse of semi-formal and formal models, Davis do not expect the actual models to be reusable as is, but rather that their presence will cause the next SRS writer to reuse the use of such modeling languages. For this to be successful, the original models should be syntactically correct.
- In cases where one actually wants to reuse the model as is, it should have a high semantic quality. For white-box reuse, the model need to be modifiable, and should also be comprehensible and comprehended, thus one need to support techniques for achieving pragmatic quality. The model should also be annotated with additional statements making it easier to find the sought for model, thus influencing the completeness relative to the purpose context of the model relative to support the need for reuse.
- Where existing models need to be compared with models developed in a separate project, social means and techniques such as model integration and conflict resolution can be useful to investigate to what extent the solutions based on the model to be reused, should be reused.

The kind of overviews as those presented by Davis have some weaknesses. Many of these have been addressed in the work by Davis, but still comparing to our framework, we see that the properties are partly overlapping. Modifiability is for instance related to redundancy, traceability, machine readability, tracedness, and that the specification is organized and cross-referenced. The problem appear partly because the list mixes goals and means to achieve these goals and because some goals are unrealistic, even impossible to reach. According to the first definition of complete, for example, a specification should include everything that the software is supposed to do. Because customers often do not know what they need and the requirements are socially constructed, you cannot hope to build a complete specification. This is addressed in our framework with the notion of feasibility. It is indirectly addressed also by Davis by giving suggestions on standards for an SRS through other of his properties, although he is not linking these up to the discussion

on completeness. Other important points when comparing the frameworks are:

- Whereas Davis' framework is largely objectivistic, built on the belief that it is possible to state true, objective requirements to a CIS, we take into account that the requirements to a CIS are constructed as part of the dialogue between the involved participants. In this light, aspects such as consistency and validation become more complex.
- We are able to discuss the relationships between the knowledge, models, and understanding of the individual participants of the modeling effort, and not only the relationships between the abstracted need of the customers and the model in the form of an SRS on an aggregated level.
- The technical areas (physical, empirical and syntactic quality) are very poorly covered by Davis. For pragmatic quality, only some of the many means described in the literature in this area are mentioned.
- Davis does not discuss aspects of social quality. For instance, Pohl [305] regard agreement as one of the three main dimensions of a requirement specification process.
- Our framework is meant to be a general framework for the assessment of quality of models in general. This is also the main weakness of it, since the discussion easily becomes rather abstract and difficult to apply in practical work. By including the properties of requirement specifications within the framework as a specialization of the framework for this specific model type as we have outlined here though, we get the better of two worlds. This is specifically apparent in the area of semantic quality, where Davis helps us to develop a much more detailed treatment.

3.13 Chapter Summary

We have in this chapter presented a framework for the understanding of quality of conceptual models. Inspired by earlier discussion on quality of conceptual models and Requirement specifications, *model quality* has been divided into seven main areas:

- Physical quality: The relationship between knowledge of those performing modeling and the model.
- Empirical quality: The relationship between the model and another model containing the same statements being somehow regarded as better through different arrangement or layout.
- Syntactic quality: The relationship between the model and the language used for modeling.
- Semantic quality: The relationship between the model and the domain of modeling.
- Perceived semantic quality: The relationship between the knowledge of the modelers and their interpretation of the model

– Pragmatic quality: The relationship between the model and the modeller's interpretation of the model.
– Social quality: The relationship between different model interpretations.

Modeling techniques, being means to achieve syntactic, semantic, pragmatic, and social quality will be discussed in the next four chapters. We will here also discuss in more detail aspects of *tool quality*, requirements to tools used in modeling to support techniques for the enhancement of model quality. One specific kind of means, using appropriate languages for the modeling task at hand has been discussed in this chapter under the banner of *language quality*, i.e. to what extent a language is appropriate to the knowledge of the modelers, the domain to be modeled, and the interpretation of the model by social and technical actors being involved i modeling. How to use the different modeling techniques being presented in this book in concert is discussed in Chap. 8 concluding the book by delving into the area of *process quality*.

In addition to be used for a framework of understanding, the quality framework can also be used for the evaluation of conceptual modeling approaches as exemplified in [25, 54, 207]. What we are able to evaluate is the potential for the modeling approaches to support the creation of conceptual models of high quality. The discussion on language quality can be used directly for evaluation purposes. An example of this is given in Appendix A.

4. Means for Achieving Syntactic Quality

We have increased the font size in Fig. 4.1 of the relationship of the quality framework that is looked into in this chapter.

The goal for syntactic quality is **syntactical correctness**, meaning that all statements in the model are according to the syntax and vocabulary of the language.

To assure the syntactic quality of the model, *syntax checks* should be provided as an integral part of the modeling support. The checks can be viewed as the simplest verification techniques and may be carried out along two main directions:

- *Error prevention*: This type of checks adapts the principles of *syntax-directed editors*. Thus, only modeling constructs that are defined in the language's vocabulary are available through the editor. Also, when a drawing session violates a syntax rule of the language, the modeling session should be temporarily interrupted in order to restore the legal model. This type of checks is controlled by the tool.
- *Error detection*: During a modeling session, some syntactical errors— *syntactic incompletenesses* — should be allowed on a temporary basis. For instance, although the DFD language requires that all processes are linked to a flow, it is difficult to draw a process and a flow simultaneously. Syntactical completeness has to be checked upon user's request. So, in contrast to implicit checks where the tool is "forcing" the user to follow the language syntax, explicit check can only detect and report on existing errors. The user has to make the corrections.

By distinguishing between these types of syntax checks, *modeling freedom* is somewhat encouraged. Throughout the modeling process, the tool will accept some syntactical errors, but these can be detected upon the user's request. The developer is free to construct the model unless syntax rules are directly violated. Modeling freedom has been discussed by Feather [113]. Although error-free models are the major goal of model quality assurance, some errors can be advantageous to have on a early in development. Too much focus on model quality at this stage might hamper the creativity of the modeling process. This idea is summed up in what is termed 'The Heisenberg Uncertainty Principle of CASE' [168]: "High levels of inconsistency and

incompleteness are permissible if they are confined to a small region of space and time."

A third syntactic mean is error correction. Error correction - to replace a detected error with a correct statement - is more difficult to automate. When implemented, it usually works as a typical spell-checker, giving hints to the correct modeling structure, but leaving it up to the modeler to do the actual change.

All syntactic means are easier if the languages used have a formal syntax. There are several ways to describe the languages for conceptual modeling, to among other things support error detection and prevention. We will discuss this in more detail below.

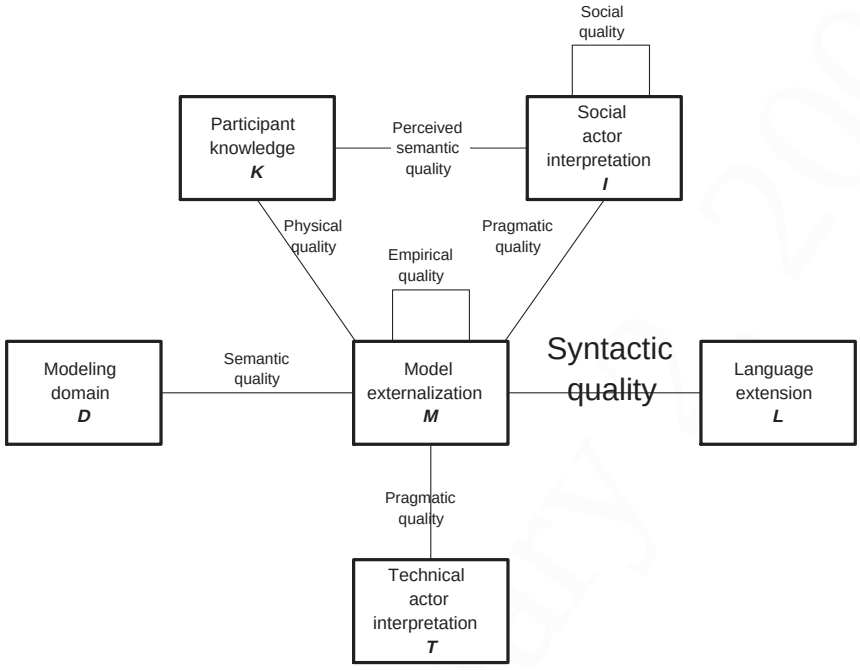


Fig. 4.1. Coverage of this chapter

4.1 Metalanguages for Syntax Specification

The expressiveness of the metalanguage used for specification of syntax and semantics determines which languages can be defined, but the metalanguage also determines the ease with which the specification is performed and understood. It has the same importance for the meta-model, as a conceptual

modeling language has for its conceptual models. There are potentially many languages suitable for syntax specification. Some of these are grouped and presented below.

Backus-Naur Form [135]. This is a widely used language for specification of programming language syntax, but can also be used for specification of conceptual modeling languages. It exists in many variants, but common to these is that they can be used to specify a class of grammars called *context-free phrase grammars (CFG)*. This is a subtype of phrase-structure grammars (Sowa [352]), which generate sentences being built from hierarchies of phrases.

CFG's are often used due to their simplicity, which makes parsing of sentences relatively easy. A weakness of CFG's is that they can not express complex constraints. An example of its use is given in Fig. 4.2.

Data modeling languages. These provide a natural way of expressing basic language constructs and their interrelationships and properties. Both Brinkkemper [40], Lyytinen [249], Sorenson [351], and many others use data modeling languages for meta-modeling. A great advantage of data modeling languages is the ease with which the language syntax is understood, at least when the data model is graphical. Also, data models often provide straightforward mappings to database schemas, which means that large parts of the storage structures for the tool supporting conceptual modeling can be easily derived. On the other hand, most data modeling languages offer limited expressiveness for constraint specification. Another disadvantage is that, although possible, data modeling languages are not that well suited to specify languages with complex phrase structures.

Predicate Logic. Logic is a powerful language. Compared to data modeling languages, it allows also complex constraints to be specified. The capabilities of logic for meta-modeling is demonstrated by Brinkkemper [40] and Falkenberg [110]. An obvious drawback of predicate logic is that it is less understandable, and also it is not well suited to specify phrase-structure grammars.

Combined Representation Languages. Some of these languages combine the expressive power of data modeling languages, restricted versions of predicate logic, and the possibility to model on the instance level. In the DAIDA project [187], Telos [269] is used as a metalanguage, offering necessary modeling constructs to be defined through meta-classes (i.e. classification-mechanisms). In the AMADEUS project [72], a frame language is used to represent a unified model of different conceptual modeling languages. Also, Sowa's conceptual graphs have been used for meta-modeling [115]. Although these languages are very expressive, they are also unsuitable for specifying phrase-structure grammars.

A Language for Representing Conceptual Modeling Languages. An executable conceptual modeling language often uses expressions in accordance with phrase-structure grammars. At the same time, their models are

often graphs with relationships between instantiated language constructs. Data modeling languages are hence suitable for specifying these constructs, their properties and relationships. We therefore propose the use of an integrated metalanguage, which combines a data modeling language with a language for specification of CFG's. In addition, we suggest use of typed first order logic when the structural constraints of the data modeling language are not sufficient.

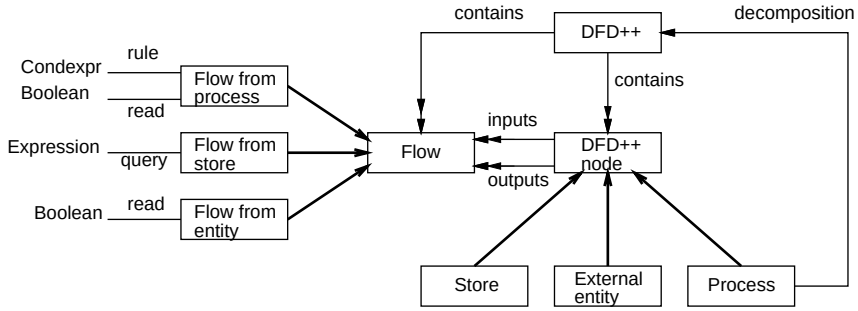
Such a data modeling language is previously depicted in Fig. 2.35.

An extended BNF can be integrated with the data model by allowing the types of construct properties to be strings generated by a particular non-terminal. In this BNF, alternative productions are indicated by '|', repeated productions (one or more, separated by commas) by {X}*, and possible productions by [X]. Terminals are indicated by the use of hyphens or boldface characters. Non-terminals placed within < > produces constants, most often text strings. We use a typed first order logic to express further constraints. Variables used in formulas range over constructs found in the meta-model, and the dot-notation is used in order to access values of construct properties. This integrated metalanguage is sufficiently expressive for specification of the syntax of executable conceptual modeling languages. It exploits the advantages of its sub-languages.

A Meta-modeling Example. As an example, we want to define an executable conceptual modeling language which is a straightforward extension of the traditional DFD. DFD++ has the same basic constructs as the traditional DFD, but here, flows are allowed to have a numeric value. Rules can be used to specify the values of output flows of processes, as a function of the input flows of the process. These are very similar to entries of decision tables, where conditions guard different alternative output expressions. Flows from stores are assumed to be queries, and these are arithmetic expressions involving the input flows to the store. Flows from external entities are simply regarded as inputs of values from the user. Flows from processes to stores result in the store being updated with the new flow value, i.e. there is implicitly a variable corresponding to the flow in the store. It is assumed that all input flows of a process must have new unread instances in order to trigger the process. A read flag indicates whether the last flow instance has been read or not, for flows from entities and processes. A flow from a store is assumed to always be available.

Portions of the meta-model for this language is depicted in Fig. 4.2. Some properties of constructs, e.g. names of processes, have been omitted. Also, some important constraints need to be added to make the meta-model complete. As an example, a requirement must be made to assure that all input flows to a store come from processes. This can be formulated in logic as follows:

$$\forall f : flow \forall s : store(f \in s.inputs \rightarrow \exists p : process(f \in p.outputs))$$




```
Condexpr        ::= '('Condition ':' Expression ';' Restcondexpr')'
Restcondexpr    ::= Condition ':' Expression ';' Restcondexpr | 'else' Expression
Condition       ::= Expression Relop Expression | Condition 'AND' Condition |
                    Condition 'OR' Condition | 'NOT(' Condition ')' | '(' Condition ')'
Expression      ::= <Flow> | Expression Op <Flow> | (Expression)
Op              ::=  '+' | '−' | 'x ' | '\'
Relop           ::=   '=' | '>' | '<'
```

Fig. 4.2. Portions of a meta-model for an executable DFD

A meta-model that is specifically geared towards the representation of executional conceptual modeling languages have been created by Willumsen [398], and extended by Krogstie [207]. This is used in connection with the execution of PPP models as will be discussed in more detail in Chap. 6.

The extended meta-model is shown in Fig. 4.3, and the main language constructs and relationships are described briefly below.

- State component: Something that can be uniquely identified, and has an associated state. Excluding executional aspects, the state of a system is the aggregate state of all state components in the system. State components correspond to entities, the static part of objects and actors, named variables etc.
- Class: A class has a name, and consists of a set of state components which have common properties.
- Attribute: A function which maps a state component to a value. The value of all properties of a state component in a class constitute the state of that state component.
- Type: An intentional description of state components. It has a name and a definition.
- Type definition: Specifies a type in terms of other types through type constructor like sets, aggregates etc.
- Condition: A logical sentence which refers to states of state components and is either true or false in a given state.

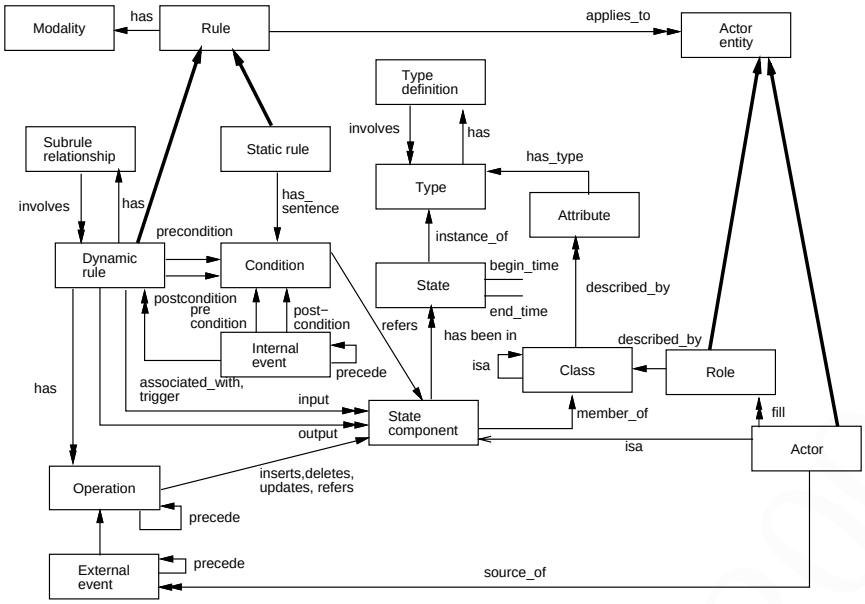


Fig. 4.3. Meta-model for execution of conceptual modeling languages

- Static rule: Includes a condition which should hold in all states. It further has a name, it specifies the situation in which the condition should be evaluated, and it specifies what action should be taken if it is violated.
- State: As defined above, but including in addition a begin-time and an end-time of the state.
- Role: Generally, behavior that can be expected by an actor by other actors. Here it is the static description of the role which is represented.
- Dynamic rule: Causes state transitions in a pre-described manner. It covers construct like process, rule, activity, method, etc.
- Sub-rule relationship: Links dynamic rules in a hierarchical rule structure, i.e. it specifies control structures between dynamic rules. It has a name, and it must have a specification.
- Operation: A specification of a state change or a query. It may insert, delete, or update state components.
- External event: A state change imposed by the environment of the system. The state change is related to two subsequent system states.
- Internal event: A state change imposed by a dynamic rule in the system.
- Actor: The initiator of external events.
- Actor entity: A generalization of role and actor.
- Rule: A generalization of state and dynamic rules.
- Modality: The modal status of a rule. Possible values are necessitation, obligation, recommendation, permission, discouragement, prohibition, and exclusion.

The relationships in the meta-model are further explained in Table 4.1.

Table 4.1. Relationships in meta-model for general execution

Relationship	Explanation
applies_to(*r*,*a*)	A rule *r* applies to a set of roles and actors *a*.
associated_with(*ie*,*dl*)	Event *ie* is caused by the execution of *dl*.
deletes(*o*,*sc*)	Operation *o* deletes a state component *sc* from a class.
described_by(*c*,*a*)	All members of class *c* have attribute *a*.
described_by(*r*,*c*)	A role *r* is described by a class *c*.
fill(*a*,*r*)	Actor *a* fill role *r*.
has_type(*a*,*t*)	The value of *a* is in the extension of *t*.
has(*t*,*td*)	Type *t* is defined through type definition *td*.
has(*dr*,*sr*)	The complex dynamic rule *dr* has subrules in rel.*sr*.
has(*dr*,*o*)	Dynamic rule *dr* is simple, and *o* executes as part of *dr*.
has(*r*,*m*)	A rule *r* has modality *m*.
has_been_in(*sc*,*s*)	State component *sc* has been or are currently in state *s*.
has_sentence(*sr*,*c*)	Static rule *sr* requires *c* to hold in every state.
input(*dr*,*sc*)	State component *s* is an input parameter to *dr*.
inserts(*o*,*s*)	Operation *o* inserts a state component *s* into a class.
instance_of(*s*,*t*)	The state *s* is in the extension of *t*.
involves(*td*,*t*)	Type definition *td* involves *t*.
involves(*sr*,*dr*)	Sub-rule relationship *sr* involves sub-rule *dr*.
isa(*c1*,*c2*)	All member of class *c1* are members of class *c2*.
member_of(*sc*,*c*)	State component *sc* is a member of class *c*.
output(*dr*,*s*)	State component *s* is an output parameter from *dr*.
precede(*o1*,*o2*)	Operation *o1* is executed before operation *o2*
precede(*ee1*,*ee2*)	External event *ee1* must happen before *ee2*.
precede(*ie1*,*ie2*)	External event *ie1* must happen before *ie2*.
precondition(*dr*,*c*)	Condition *c* must hold for *dr* to apply.
precondition(*ie*,*c*)	Condition *c* holds in the state when *ie* happens.
postcondition(*ie*,*c*)	Condition *c* holds in the state when *ie* has happened.
postcondition(*dr*,*c*)	Condition *c* must hold in the state *dr* terminates.
refers(*o*,*sc*)	Operation *o* refers to a state component *sc* in a class.
refers(*c*,*sc*)	Condition *c* involves reference to a state component *sc*.
source_of(a,*ee*)	The source of an external event *ee* is actor *a*.
specified_by(*ee*,*o*)	Operation *o* is performed when *ee* is reported.
trigger(*ie*,*dr*)	The event *ie* may trigger the execution of *dr*.
updates(*o*,*sc*)	Operation *o* updates a state component *sc* in a class.

One can translate models written in the PPP languages and models written in other languages such as Petri-nets and SQL into this language for further execution. We will return to execution of conceptual models in Chap. 6.

Another use of language models is for so-called meta-models which are a starting point for the repository support of a modeling tool. We use parts of the model for PPP as example. The model is written using ONER, the data modeling language of PPP.

Figure 4.4 shows a ONER-scenario which includes the meta-model in connection to the rule and process modeling languages.

The main entities are:

– DRL-rule: A DRL-rule has an ID, a type (i.e. constraint, derivation rule, action rule, predicate, non-functional or undefined), and a short-name. The rule can be stated as a formal DRL-rule or as a natural language (NL) expression or both. The DRL-rule can be further divided into the trigger, condition (which can refer to a scenario of a ONER-model) and a (set of) consequences. This is not indicated in the figure. In addition it can have one of the five deontic operators, and can apply to a role or an actor. It is also possible to capture which actor that has institutionalized the rule.

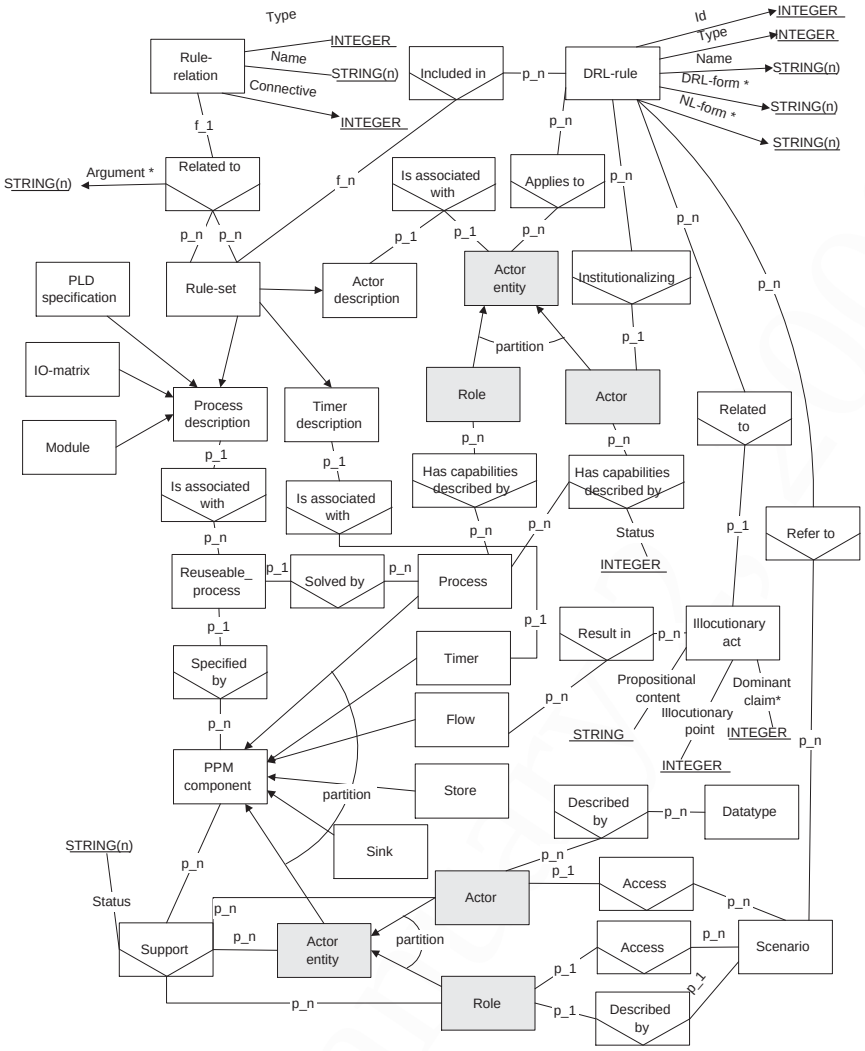


Fig. 4.4. Incorporating the use of rules and actors in the PPP meta-model

- Rule-set: The clustering of rules into rule-sets is necessary for general rule-relations to be specified. The *rule-relation* has a type, indicating the type of relation (necessitate, obligates etc), a name, and a connective (and/or/none). The related-to relationship links a relation with two rule-sets, and can have an optional argument.
- The capabilities of actors and roles are described by processes. For the capabilities of an actor it is also possible to indicate the status, i.e. if they are proven or only potential. An actor or role can be said to support a set of PPM-components. An actor can access data described in a scenario of a data model, and can be described by a set of data-types in addition to the ones indicated through the link between the roles they fill, and how these roles are described statically in the ONER-model. If an actor support a store, it is implicitly given that the actor can access the data described in the scenario for the store. An actor-entity (actor or role) can finally be associated with a set of rules, used in model simulations.
- PPM components can be process, timer, flow, store, sink, actor, and role. Processes and timers are potentially associated with a process description in the form of a rule-set, a PLD, or an i/o-matrix. Flows are optionally connected to a set of illocutionary acts, for which one can indicate the propositional content, the illocutionary point, and the dominant claim. An illocutionary act can further be related to one or more rules.

An example of more detailed rules for ports in PPM is given below:

S*1. Every process has at least one CIP and at least one COP;*

S*2. A CIP has at least one triggering input, and a COP has at least one terminating output;*

S*3. The i/o condition of a process must ensure that every CIP satisfies at least one COP, meanwhile any COP must be satisfied by at least one CIP.*

We say that a CIP satisfies a COP if and only if

In the CIP, either all of the triggering non-conditional inputs (if any) or any of the conditional triggering inputs, along with all non-triggering and non-conditional inputs (if any), satisfy the necessary conditions for all non-conditional outputs in the COP. Furthermore, if there is no terminating and non-conditional output in the COP, then those inputs must satisfy at least one conditional and terminating output in the COP;

S*4. The i/o condition of a process must not include a conjunction term of inputs that can not be in any CIP of the process;*

S*5. The set of the members of a canonical port can not be a subset of another canonical port;*

S*6. Only a singular input/output can be a triggering/terminating input/output;*

S*7. If in the i/o condition of a process all conjunction terms of an output contains at least one conditional input, then the output must also be a conditional output;*

S*8. If a group of inputs are the triggering inputs of a CIP for the higher level processes, then in the process network that is the result of a decomposition of*

the process, they must also be the triggering inputs of a CIP for one sub-process in the network; if a group of outputs are the terminating outputs of a COP for the processes, then in the process network , they must also be the terminating outputs of a COP for one sub-process in the network.

Error prevention is supported by the PPP tool. The PPP editors only provide the modeling constructs that are defined in the PPP language. Thus, syntactic invalidity in the PPP model are avoided. Moreover, the PPP editors resemble *syntax-directed editors*. Drawing sessions that violate the language grammar are disrupted in order to restore the legal model.

In addition to this implicit syntax checking of PPM model, the tool provides checks for syntactical completeness — the model is temporarily lacking constructs with respect to the PPM syntax. These checks can be invoked by the user. Thus, they are explicitly supported by the tool.

In Fig. 4.5 a PPM model is shown. At this stage the model is still syntactically incomplete. For instance, process P1 is lacking output flows, whereas actor A1 is lacking a name. By checking the syntactical completeness of the model, the tool will respond with error messages shown in the figure.

As indicated in Fig. 4.5, the scope of checks can be decided by the user. In addition to flow completeness and object naming, the model can also be checked for (1) triggering/terminating flow properties, (2) input and output ports, (3) flow and store content, and (4) decomposition of processes. Since the modeling session has not yet addressed these issues, such checks are rather unnecessary at this stage, though.

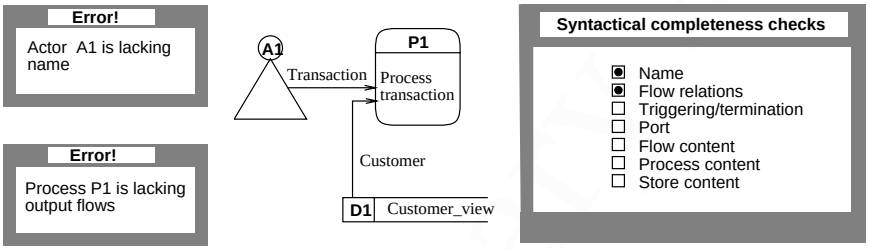


Fig. 4.5. Syntactical completeness checks of PPM model

4.2 Chapter Summary

Syntactic quality is the correspondence between the model $\mathcal{M}$ and the language extension $\mathcal{L}$ of the language in which the model is written. There is only one syntactic goal, **syntactical correctness**, meaning that all statements in the model are according to the syntax and vocabulary of the language.

Syntax errors are of two kinds:**Syntactic invalidity**, in which words or graphemes not part of the language are used and **syntactic incompleteness**, in which the model lacks constructs or information to obey the language's grammar.

Syntax checks can be viewed as the simplest verification techniques and may be carried out along three directions: *Error prevention, error detection, error correction.* Although simple, as has been shown in Chap. 3.12, semantic problems might be transferred into syntactic problems by extending the language in appropriate ways on the danger of closure. All syntactic means are easier if the languages used have a formal syntax, and we have in the chapter indicated different ways of describing language for conceptual modeling formally.

5. Means for Achieving Semantic Quality

We have increased the font size in Fig. 5.1 of the relationships of the quality framework that are looked into in this chapter.

Semantic quality is the correspondence between the model and the modeling domain. The framework contains two semantic goals; validity and completeness. *Perceived* semantic quality is similarly correspondence between the actor interpretation of a model and his or hers current knowledge of the domain.

Activities for establishing higher semantic quality, are statement insertion and deletion. An update is a deletion followed by an insertion. Statement insertions and deletions can obviously result in lower semantic quality. Statement insertion and deletion can generally be looked upon as meaning updating transformations, which can be done either manually or automatically. An examples of the latter is the situation with a syntactically complete delete as discussed in Chap. 3. Other examples are the evolution transformations described in [189], and work within goal-oriented modeling [270]. Of specific importance is direct model reuse (being a specific type of statement insertion). This can either be the reuse of a previous model of a similar domain, or might be a translation of a previously baselined model.

Consistency checking is another activity here. To be able to do consistency checking, the model must be made in a formal, preferably logical language, and to enable and assess the impact of updates, it should be modifiable. This includes properties such as structure, locality of changes, and control of redundancy. Consistency checking can be looked upon as one of several types of model testing which is beneficial at this level.

A wide range of modern conceptual modeling techniques start out by verbalizing sample forms, cases and so on [368]. The verbalizations resulting from this step are then used for the development of the first version of the model. A technique for further elaboration on the model is the use of driving questions based on the already existing model as used in Tempora [389]. For feasibility analysis, however, there is little in way of tool support, except when the domain is already described in another model, or a standard has been agreed upon. General tools are difficult to develop for the simple reason that the domain and audience are beyond automatic manipulation. The means for achieving a high perceived validity and completeness are similar to the ones

for traditional validity and completeness, with the addition of participant training.

Using a formal language one can in a sense translate a semantic problem into a syntactic one, but this sets additional requirements to the domain appropriateness of the language.

We will in this chapter discuss the following techniques in more detail:

- Consistency checking based on a logical description.
- Consistency checking based on constructivity.
- The use of driving questions to improve completeness.

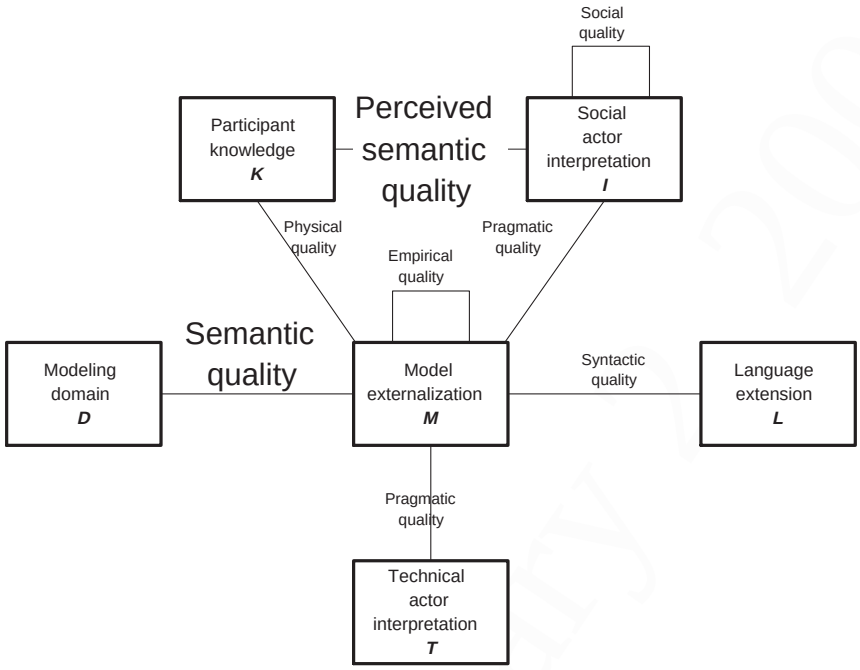


Fig. 5.1. Coverage of this chapter

5.1 Consistency Checking

There are two main approaches for formal specifications as the basis of consistency verification:

- The *algebraic specification approach* specify a system as a set of abstract data types (ADTs). The theory of an ADT consists of a set of symbols

(sorts and operations) (signature) and a collection of formulae (the axioms of the theory); the interpretation of the theory is a many sorted algebra [39, 101, 322, 402]. The specification is a set of theories and the relationships between them. This approach has been applied in the object-oriented systems [116, 196, 333, 334, 335, 394];
- The *logical theory approach*, on the other side, treat the schema of a database and the associated operations on it as one logical theory. This approach has also been applied widely [128, 182, 282, 320].

Both approaches have advantages. Here we present the logical approach. The logical view supports the idea of a *global state of a database* [202]. Kung [218] has provided a set of methods for consistency verifications on information system specifications including static constraints, operation specifications and temporal constraints along with the algorithms to check if the specification is decidable with respect to its consistence.

One can not expect to express all parts of a conceptual model as one logical theory and use the logical inference mechanism to verify the consistency of the system model.

5.1.1 Formal Verification of Data Models

It has been shown in [404] how a semantic data model such as one written in ONER can be translated into a logical theory, i.e. a first order and many sorted logical theory [1]. In turn the consistency of the theory can be verified formally by means of logical inferences.

Many first order theories are only semi-decidable, i.e., when they are inconsistent, the inconsistency can be found within finite steps of logical inferences. However, if they are consistent but not logically valid, i.e., there are some but not all interpretations on which the theories are true, then there is no effective method to prove the consistency within finite number of steps.

The verification method for ONER is based on the work of Lewis and Kung [218, 217, 230, 231], in which they defined the weakest condition to determine that if the consistency of a set of clauses is decidable, and proposed the methods to check the condition by means of graph theory. In addition, the consistency of some decidable clauses can be conducted directly, whereas other decidable clauses have to be checked by resolution.

Lewis' work. Generally speaking, the consistency of a theory, i.e., a set W of Well-formed formulaes ($wffs$), is undecidable. However, if certain restrictions are imposed upon W, then the consistency of W can be effectively determined. The problem is that, if the restrictions are too strong, then many interesting clauses of $wffs$ will be excluded. A practical method must let its restrictions be as weak as possible.

[1] In a many sorted logic the universe of discourse is divided up into subsets and variables are restricted to range over these subsets, called sorts, rather than the whole universe as is classically the case

A weak condition for restricting a set W of wffs is found in Lewis' work [230]. This condition is defined in terms of the clause form of W. A wff is said to be in clause form if it is of the form

$\alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_n$

where α_i $(i = 1, \cdots, n)$ is either an atomic formula or the negation of an atomic formula. α_i is also called a prime formula or a *literal*.

Definition 5.1.1.1: Let S be a set of clauses. An S-link is an ordered triple $< C, \alpha, \beta >$, where C is a clause and α, β are distinct literals in C. An S-chain is a sequence $< C_1, \alpha_1, \beta_1 >, \cdots, < C_n, \alpha_n, \beta_n >$ of S-links such that for $i = 1, \cdots, n-1$, β_i is unifiable with $\sim \alpha_{i+1}$; this S-link is said to have length n. If β_n is unifiable with $\sim \alpha_1$, then this S-chain is called an S-cycle.

For example, let S consists of the following set of clauses

$\sim A_1(x_1) \vee A_2(x_1, f_1(x_1))$

$\sim A_1(x_2) \vee A_3(f_2(x_2), a_1)$

$\sim A_2(x_3, y_3) \vee A_4(y_3)$

$\sim A_4(y_4) \vee \sim A_3(f_2(y_4), a_1)$

According to definition 5.1.1.1, S has the following 8 S-links, since the example has 4 clauses each of which has two literals:

$< C_1, \sim A_1(x_1), A_2(x_1, f_1(x_1)) >$

$< C_1, A_2(x_1, f_1(x_1)), \sim A_1(x_1) >$

$< C_2, \sim A_1(x_2), A_3(f_2(x_2, a_1) >$

$< C_2, A_3(f_2(x_2, a_1), \sim A_1(x_2) >$

$< C_3, \sim A_2(x_3, y_3), A_4(y_3) >$

$< C_3, A_4(y_3), \sim A_2(x_3, y_3) >$

$< C_4, \sim A_4(y_4), \sim A_3(f_2(y_4), a_1) >$

$< C_4, \sim A_3(f_2(y_4), a_1), \sim A_4(y_4) >$

One of the longest S-chains of the example is of length 4 as shown below:

$< C_1, \sim A_1(x_1), A_2(x_1, f_1(x_1)) >$

$< C_3, \sim A_2(x_3, y_3), A_4(y_3) >$

$< C_4, \sim A_4(y_4), \sim A_3(f_2(y_4), a_1) >$

$< C_2, A_3(f_2(x_2, a_1), \sim A_1(x_2) >$

If we start from another clause, then we can have another S-chain. However, it can be verified that the example contains no S-cycle.

Definition 5.1.1.2: A set S of clauses is compact if S contains no S-cycle.

Theorem 5.1.1.1: Satisfiability is decidable for compact sets of clauses.

The proof of the theorem is given in [230].

Theorem 5.1.1.1 means that if a set S of clauses is compact, it can be determined by logical inference within a finite number of steps if it is consis-

tent. Moreover, Lewis has also proved that compact sets of clauses are the *maximum sets* for which consistency is decidable. In other words, the above definitions determines the weakest condition for a theory to be decidable.

Kung's work. Kung has given a definition of compactness which is equivalent with that of Lewis' but in terms of graph theory. Using graph theory, it is simpler to check the compactness of a set S of clauses.

Before introducing the definitions and theorems of Kung's work, we need a brief description of some terms for *directed graphs*.

A directed graph or digraph G is an ordered pair $< V, E >$, where V is a finite set of nodes and $E \in V \times V$ is a finite set of edges. $< u, v >$ means that the edge goes from u to v.

The *outdegree* of a node is the number of edges going from it. The *indegree* of a node is the number of edges coming into it.

A *path* is a sequence $v_0, \cdots, v_n$ of nodes ($n \geq 0$) such that $< v_i, v_{i+1} >$ is an edge for $i = 0, \cdots, n-1$, such that the v_is are distinct. Except that we allow $n > 0$ and $v_0 = v_n$, in which case the path is a *cycle*. A digraph with no cycle is *acyclic*. If there is a path from u to v, then v is *reachable* from u. u is reachable from u for all u.

Semipath and *semicycle* are defined like path and cycle, except that either $< v_i, v_{i+1} >$ or $< v_{i+1}, v_i >$ may be an edge.

A digraph is *weakly connected* if there is a semipath between every pair of nodes. A digraph which is not weakly connected consists of *weak components* each of which is *maximally weakly connected*. A digraph is strongly connected if every two nodes are mutually reachable.

Definition 5.1.1.3: A *unifiable digraph* $G(S) = < S, E >$ for a set S of clauses as defined as follows.

1. $G = < S, E >$ has S as its nodes and $E \subseteq S \times S$ as its set of edges;
2. $< C_i, C_j >$ belongs to E if for some positive $\alpha \in C_i$, negative $\beta \in C_j$, such that $\sim \alpha$ is unifiable with β for some unifier θ. In this case, we label the edge by $C_i.\alpha / C_j.\beta + \theta$. ($i$ and j may be equal).

Definition 5.1.1.4: Let $G(S)$ be a unifiable digraph of S. Two edges

$e_1 = C_{i_1}.\alpha_1 / C_{j_1}.\beta_1 + \theta_1$ and

$e_1 = C_{i_2}.\alpha_2 / C_{j_1}.\beta_2 + \theta_2$

which belong to $G(S)$ are said to be *conflicting* if

$(C_{i_1} = C_{i_2} \wedge \alpha_1 = \alpha_2) \vee (C_{j_1} = C_{j_2} \wedge \beta_1 = \beta_2)$

Definition 5.1.1.5: A semicycle of $G(S)$ is said to be conflicting if it contains a node C together with a pair of its conflicting degrees.

Definition 5.1.1.6: A unifiable digraph $G(S)$ is said to be *compact* if $G(S)$ contains only conflicting semicycles (if any).

As an example, let S be the set of clauses:

C_1: $\sim WF(x_1, a) \vee WF(b, x_1)$

C_1: $\sim WF(x_2, x_2) \vee \sim WF(y_2, x_2)$

C_3: $WF(b, a)$
then $G(S)$ is compact in Fig. 5.2.

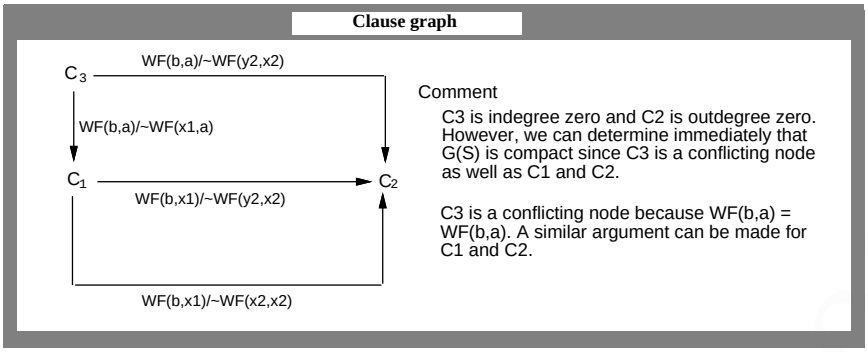


Fig. 5.2. A compact unifiability digraph

Based on the proof of that S is compact (Lewis' definition) if and only if $G(S)$ is compact (Kung's definition), Kung gives the following theorem:

Theorem 5.1.1.2: Satisfiability is decidable for a set S of clauses if $G(S)$ is compact.

The theorem provides us the capability of checking if a theory is decidable by means of graph theory, when it is transformed into a set of clauses. Furthermore, Kung has proved other theorems on the properties of $G(S)$ that may provide more information about the clause; for instance, the satisfiability of some clause sets can be determined directly under special conditions.

Theorem 5.1.1.3: Suppose that $G(S)$ is not weakly connected, and let $< S_1, E_1 >, < S_2, E_2 >, \cdots, < S_n, E_n >$ be the weak components of $G(S)$. Then S is satisfiable *iff* each of $S_1, \cdots, S_n$ is satisfiable.

Theorem 5.1.1.4: If $G(S)$ contains no node of indegree zero, or no node of outdegree zero, then S is satisfiable.

Theorem 5.1.1.5: If $G(S)$ is strongly connected then S is satisfiable. In particular, if $G(S)$ consists of only one node, then then S is satisfiable.

Theorem 5.1.1.6: if $G(S)$ consists of only a single non-conflicting semicycle, then S is consistent.

A node C is called a *redundant node* if C contains a literal α which is not unifiable with the negation of any other literals in S. $G(S)$ is called redundant if it contains a redundant node.

Theorem 5.1.1.7: S is consistent if and only if $S - \{C_i\}$ is consistent, where C_i is a redundant node of $G(S)$.

The Consistency Verification for ONER. On the basis of [404] and Lewis' and Kung's work, one can define the following steps to check the consistency of a database schema:

1. Collect the concepts and constraints from one or more ONER scenario that are considered to be the conceptual schema of a database.
2. Form the formal DB schema from this scenario.
3. Transform the DB schema into a many sorted and first order logical theory.
4. Transform the theory into a set of clauses S. The method of transforming is the standard 7-step method that is given in e.g. [134].
5. Construct the digraph $G(S)$ as defined in Sect. 5.1.1.
6. Check $G(S)$ to see if it is weakly connected. If it is, then rename S to S_1, else identify the weakly connected sets $S_1, \cdots, S_n$ each of which is *maximally weakly connected.*
7. For each S_i $(i = 1, \cdots, n)$ do the following:
   - By the theorems in Sect. 5.1.1, check if $G(S_i)$ can be determined to be consistent directly;
   - If $G(S_i)$ can not be determined to be consistent directly, then use the theorem 5.1.1.2 to check whether $G(S_i)$ is decidable;
   - If $G(S_i)$ is decidable, then use the *resolution* method to check its consistency. The method of resolution is given in many books, e.g. [134]. Since S_i is decidable, the resolution process will end within a finite number of steps.
8. If every S_i $(i = 1, \cdots, n)$ is consistent, then S is proven to be consistent; if any S_i is inconsistent, then S is proven to be inconsistent; if any S_i is undecidable, then S is undecidable.

This algorithm will produce one of three possible results about the consistency of the DB schema: 1) It is consistent; 2) It is inconsistent; 3) The consistency of the schema is undecidable, i.e., can not be checked with an effective algorithm within finite steps.

If the result is the third case, it does not mean that the consistency can not be checked. We know that many first order theories are semi-decidable, i.e., if a theory is inconsistent, this can be checked within a finite number of steps. Therefore, even though a set S of clauses is undecidable, we can still try to use the resolution method to check it. If after a large number of steps the process still do not stop, then we can remind the users about undecidability, and suggest that they try to construct an example themselves. If an example (model) can be constructed, where all the formulae of the theory are satisfiable, then the schema is still proven to be consistent.

The proofs of the theorems presented here can be found in [218, 230].

5.1.2 Static Consistency Checking for PPM

The *i/o conditions* of the PPM can be calculated without considering the dynamic executions. For example, Tao and Kung proposed to calculate the *transitive closure of the precedence relation* of a DFD [364] in order to check the correctness of a decomposition with respect to the precedence relations between the inputs and outputs of a decomposed process. In PPM, differently, we use *i/o conditions* that provide more accurate information about the relations between an output and some of the inputs of a process. We thus use a *substitution* algorithm to calculate the *i/o condition expression* of an output in the PPM which is the decomposition of a process, with the similar principle to that of Tao and Kung.

This algorithm is used for each output of a process network to conduct its i/o condition expression on the external inputs to the network. First the condition expression(s) for the output of the sub-process(es) is taken or merged into a "current expression", then the input names in the expressions are substituted with the condition expressions for the outputs of other processes that are linked with the inputs through flow-pipes in the process network, while those parts in the expression with illegal combination of external inputs are deleted. The substitution is done recursively until the current expression can not be changed any more. After deleting the terms containing internal outputs produced in some loops of the network, the final expression is just the i/o condition expression of the output on the process network. If the result is empty, then the process network is inconsistent because an output can not get the data needed from the outside of the network during an execution of the process network. The result should also be compared with the i/o condition of the decomposed process. The detailed algorithm is given in Appendix B.

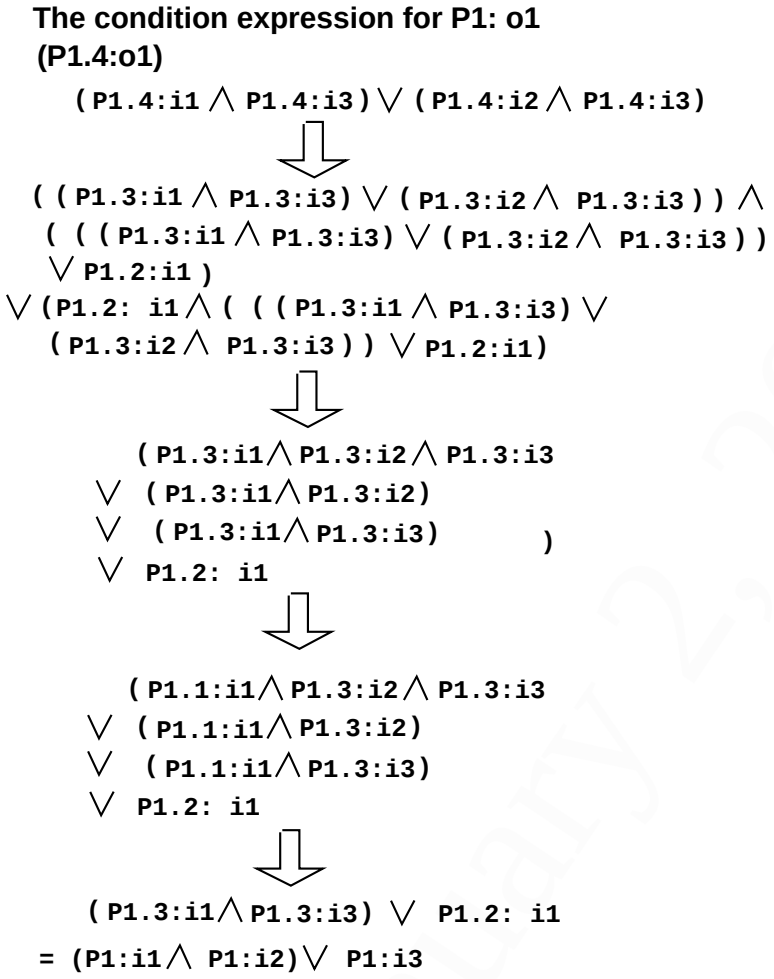


Fig. 5.3. The i/o condition for the process network for P_1 of the IFIP ticket booking activities

In Fig. 5.3, we calculate the condition expression for the outputs o_1 and o_2 of P_1 in the process network shown in Fig. 2.45. The result is same as the original i/o condition of P_1, thus we can conclude that the decomposition is correct on the aspect of i/o conditions.

5.2 Constructivity – The Fundamental Principle

The notion of constructivity was brought into the field of information systems engineering by Langefors [223], who defined what he called *the fundamental principle of systems work*. This principle amounted to the following:

> Partition the systems work into separate parts, **a** through **d**:
> a. Definition of the system as a set of parts:
> List all parts from which the system is regarded as built-up.
> b. Definition of the system structure:
> Define all interconnections which make up the system by joining its parts together.
> c. Definition of the systems parts:
> For each single part (or group of similar parts) separately define its properties as required by the system work at hand and do this in a format as specified by the way the systems structure is defined (in task b).
> d. Determination of the properties of the system:
> Use the definitions as produced by the tasks a, b, and all separate tasks c, all taken together. Compare with specifications wanted for the system and repeat a, b, c, and d until satisfied.

It is the point **d** in the list above that involves constructivity. Basically, it means to derive the properties of a system based on the properties of its subsystems, and then check if the derived properties are the same as those specified for the system earlier (if any). The derivation can be called *abstraction*, and a specific subsystem structure is said to be constructive if such an abstraction is possible. Constructivity is necessary when we want to check the consistency of a hierarchical specification, i.e. to check whether decompositions are correct.

Some other important points made by Langefors are the following:

- The principle divides tasks into two sets: one concerned with the whole system (a, b, d) and one concerned with its parts (c, where each part can be treated separately).
- It gives a natural way for dividing the work among several people (which is essential for large, complex systems).
- The principle works equally well for top-down and bottom-up development.
- The principle is most efficient when it can be mathematically formalized, but one should stick to it also when this is not possible.

Although the ideas of Langefors are more than 20 years old, little has happened when it comes to automating the fundamental principle in commercial information systems development tools. Considering the potential such an automation would have, it seems natural to believe that the main reason for this is the difficulty of the task. Even though the criticism raised concerning the applicability of the fundamental principle may be correct, this does not mean that constructivity is not a nice feature to have. Obviously,

- Languages and methods that tend to give constructive subsystem structures are highly preferable to those that do not, and
- Any serious approach based on hierarchical decomposition should have some automated support for constructivity built into its specification tools.

For a more concrete illustration of the possibilities and problems connected to constructivity, we summarize some experiences from previous research in the next section.

5.2.1 Constructivity in BNM

An algorithm for abstracting a network of BNM transitions to one higher level transition whose pre- and postconditions would be the conditions of the network as a whole (i.e. performing Langefors' step d above) was outlined by Kung in 1986 and elaborated further in [342].

The algorithm goes through these three main steps:

1. Find all possible transition sequences.
2. Propagate variable updates along each such sequence to eliminate variables of internal places from the expression of external output values.
3. Build higher level network based on 1 and 2.

To illustrate the algorithm in some more detail, we give an example. The network of Fig. 5.4(a) describes the detailed behavior for handling a batch of orders. The places in the network hold the following variables:

P1 holds variable m (messages)
P2 holds variable W (orders)
P3 holds variable X (individual order being processed)
P4 holds variable P (parts)
P5 holds variable y (rest orders)
P6 holds variable z (shipped orders)

For the sake of simplicity we have not shown the attributes of the phenomenon classes. These are indicated by the ordinary point notation following variables in the conditions of the transitions:

t1: pre: m = "Handle Orders"
post: $\circ$X = W
t2: pre: $X = \emptyset$

post:

t3: pre: $(\exists x)\neg(\exists s)(x \in X \wedge\ s \in S \wedge\ x.P\# = s.P\#\ \wedge\ x.Q \leq s.Q)$
post: $\circ X = X \setminus \{x\}\ \wedge\ z.C\# = x.C\#\ \wedge\ z.P\# = x.P\#\ \wedge\ z.Q = x.Q$

t4: pre: $(\exists x)(\exists s)(x \in X \wedge\ s \in S \wedge\ x.P\# = s.P\#\ \wedge\ x.Q \leq s.Q)$
post: $\circ X = X \setminus \{x\}\ \wedge\ \circ s.Q = s.Q - x.Q \wedge\ y.C\# = x.C\#\ \wedge\ y.P\# = x.P\#\ \wedge\ y.Q = x.Q$

The network with its conditions can be explained as follows:

- When the message "Handle Orders" is given, the list of orders to be processed is copied from the variable W to the variable X (t1 does not consume P2 because P2 is only a reference place, as indicated by the dotted line).
- As long as there are still orders to process, an arbitrary order will be picked and processed. If it can be satisfied (i.e. the requested part exists and the quantity in the store is sufficient), then t3 will fire, creating an entry for the order in the pack list. Otherwise t4 will fire, creating a rest order. Both t3 and t4 take orders away from the set X one by one ($\circ X = X \setminus \{x\}$, where the circle is the temporal next operator, i.e. X in the next state is equal to X in the previous state minus the element x).
- When X finally becomes empty, t2 will fire. The firing of t2 removes the token from P3, and consequently, nothing more can happen in the network until a new "Handle Orders" message arrives, i.e. execution halts.

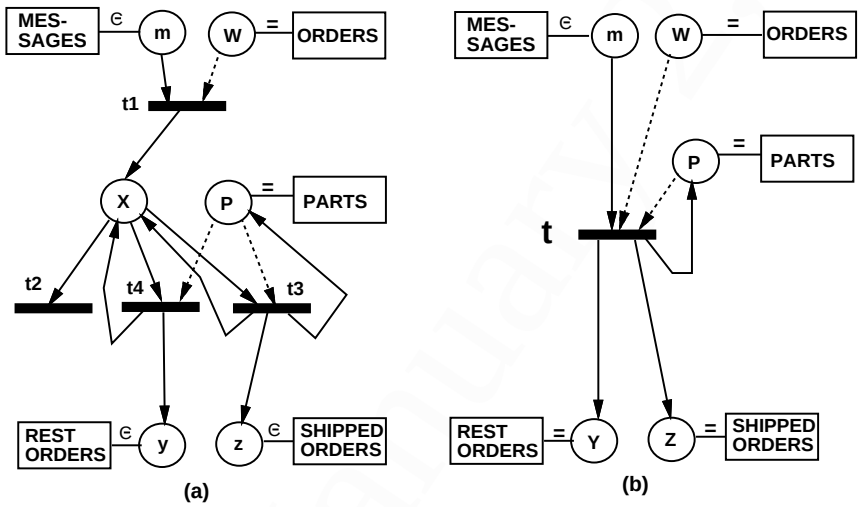


Fig. 5.4. A Behavior Network, before and after abstraction

Recognizing that the states of the two output places P5 and P6 have no impact on what can happen next in the network, we can derive automatically

the state transition diagram of Fig. 5.5. In the vector $(P1, P2, P3, P4)$ of fig.5.5

$P_i = 1$ means that the corresponding variable exists and has a value
$P_i = 0$ means that the corresponding variable has no value

Concentrating on one execution of the network (i.e. not going around and around the dotted arc resulting from new "Handle Orders" messages), we can establish that there are basically four different ways of going through this graph from top to bottom: either going directly through it (i.e. skipping both loops), including only the t3 loop (all orders can be satisfied), including only the t4 loop (no orders can be satisfied), or including both loops (satisfy some orders, others not).

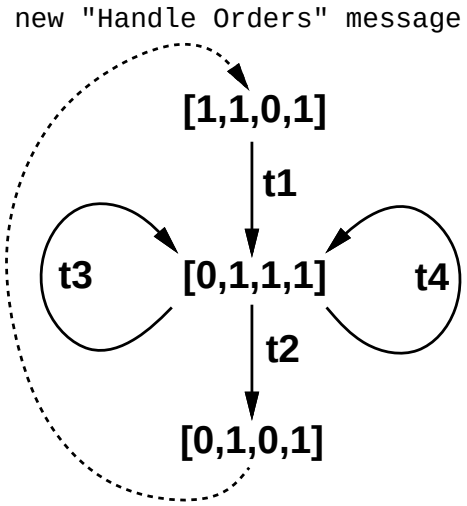


Fig. 5.5. An STD for the network

The next step of the algorithm amounts to looking at each of these possible paths through the state-transition graph in order to eliminate internal variables. This analysis is performed by using a table with 3 columns, **PC** (path condition), **SC** (state change), and **PA** (path assertion). The first will contain preconditions for the transitions of the path, the second state changes on internal variables, and the third the assertion of external output. For the alternative where both loops are skipped the analysis is very simple, as illustrated in Table 5.1.

The main clue of the path analysis is illustrated in the PC entry of t2 here. The precondition of t2 has been given as X = ∅. However, before we get to t2 in the transition sequence, we have executed t1, for which we have

Table 5.1. The analysis of the path without loops

	PC	SC	PA
t1	m = "H.O."	X = W	
t2	W = $\emptyset$		STOP

recorded the state change X = W. Whenever new conditions or state changes are to be inserted, we have to consult the last state change of all variables involved. Thus, we find that X in the precondition of t2 can be replaced with W, so that the precondition of t2 becomes W = $\emptyset$. Thus, we have managed to eliminate the internal variable X.

The analysis of the path with only the t3 loop is a little more complex. Since the number of orders must be regarded as finite, but arbitrary we first establish an expression for the first run-through of the loop, and then we extrapolate this to final run-through (the n-th). The x'es are thus indexed according to their completely arbitrary order of processing. In Table 5.2 we have used the following abbreviations:

- $U = \bigcup_{i=1}^{n-1}\{x_i\}$,
- $V = \bigcup_{i=1}^{n}\{x_i\}$,
- $M = \sum_{i=1}^{n-1} x_i.Q$,
- $N = \sum_{i=1}^{n} x_i.Q$, and
- $... = y_1.C\# = x_1.C\# \wedge y_1.P\# = x_1.P\# \wedge y_1.Q = x_1.Q$ and $y_n.C\# = x_n.C\# \wedge y_n.P\# = x_n.P\# \wedge y_n.Q = x_n.Q$, respectively.

Table 5.2. The analysis of the path with only the t3-loop

	PC	SC	PA
t1	m = "H.O."	X = W	
1st t3	$(\exists x_1)(\exists s)$	$X = W \setminus \{x_1\}$	$(\exists y_1)(...)$
	$(x_1 \in W \wedge s \in S \wedge$	$\wedge$	
	$x_1.P\# = s.P\# \wedge$	$s.Q = s.Q - x_1.Q$	
	$x_1.Q \leq s.Q)$		
nth t3	$(\exists x_n)(\exists s)$	$X = W \setminus V$	$(\exists y_n)(...)$
	$(x_n \in W \setminus U \wedge s \in S \wedge$	$\wedge$	
	$x_n.P\# = s.P\# \wedge$	$s.Q = s.Q - N$	
	$x_n.Q \leq s.Q - M)$		
t2	$W \setminus V = \emptyset$		STOP

The analysis for the other single loop path is very similar to the one just considered, only a little bit simpler (since we satisfy no orders, nothing is subtracted from the store). Thus, we do not bother to show it. The double loop path, on the other hand, is rather problematic. The state change on X can be extrapolated as easily as before, since t3 and t4 update this variable in the same way. For s.Q, on the other hand, the state change cannot be

extrapolated exactly. Whenever an x_i is taken by t3, $x_i.Q$ is subtracted from s.Q, whereas nothing is subtracted when it is taken by t4. Since the choice between t3 and t4 is itself dependent on the value of s.Q, an exact expression for how many x_i are satisfied and how many are not cannot be established, neither can we give an exact expression for the value of s.Q when the execution of the network halts – all we can say is that it must be somewhere between the old value and 0. The analysis is shown in Table 5.3.

Table 5.3. The analysis of the path with both loops

	PC	SC	PA
1st t4	$(\exists x_1)(\exists s)$ $(x_1 \in W \wedge s \in S \wedge$ $x_1.P\# = s.P\# \wedge$ $x_1.Q \leq s.Q)$	$X = W \setminus \{x_1\}$ $\wedge$ $s.Q = s.Q - x_1.Q$	$(\exists y_1)(...)$
or t3	$(\exists x_1)\neg(\exists s)$ $(x_1 \in W \wedge s \in S \wedge$ $x_1.P\# = s.P\# \wedge$ $x_1.Q \leq s.Q)$	$X = W \setminus \{x_1\}$	$(\exists y_1)(...)$
nth t4	$(\exists x_n)(\exists s)$ $(x_n \in W \setminus U \wedge s \in S \wedge$ $x_n.P\# = s.P\# \wedge$ $x_n.Q \leq s.Q-?)$	$X = W \setminus V$ $\wedge$ $s.Q = s.Q-?$	$(\exists y_n)(...)$
t2	$W \setminus V = \emptyset$		STOP
or t3	$(\exists x_n)\neg(\exists s)$ $(x_n \in W \setminus U \wedge s \in S \wedge$ $x_n.P\# = s.P\# \wedge$ $x_n.Q \leq s.Q-?)$	$X = W \setminus V$ $\wedge$	$(\exists y_n)(...)$
t2	$W \setminus V = \emptyset$		STOP

The final step is to establish the higher level network with pre- and post-condition. By some symbolic manipulation, we are able to obtain the following formulae:

$$(\forall x)(x \in W \rightarrow ((\exists y)(...) \vee (\exists z)(...)$$

and

$$\begin{aligned}&(\forall s)(s \in S \rightarrow \\ &\quad (\textstyle\sum_{x \in W \wedge x.P\# = s.P\#} x.Q \leq s.Q \\ &\qquad \rightarrow \circ s.Q = s.Q - \textstyle\sum_{x \in W \wedge x.P\# = s.P\#} x.Q) \\ &\quad \wedge (\textstyle\sum_{x \in W \wedge x.P\# = s.P\#} x.Q \leq s.Q \\ &\qquad \rightarrow \circ s.Q \in [0, s.Q]))\end{aligned}$$

for the general case where both loops are taken. Since these expressions also cover the more special cases where one or both loops are omitted, it can be used as the postcondition for the higher level transition t shown in Fig. 5.4(b).

The precondition of this higher level network will be the same as the precondition of the first transition executed, i.e. m = "Handle Orders".

As can be seen, there is a certain vagueness in the expressions, due to the problems just mentioned with the double loop path. Still, the expressions obtained are quite informative about what the network does, so evidently it is possible to do some consistency checking even in cases where we cannot express the outcome of the execution of a network exactly.

The work accounted for in [342] resulted in the implementation of a Prolog prototype for BNM abstraction, sophisticated enough to handle cases such as the order handling problem above.

5.2.2 Constructivity in PPM

In this section, we present the method for dynamic consistency checking for PPM. More specifically, it is a method to check the constructivity property of the decompositions by calculating the possible execution sequences of a process network.

In the sequel, we will first introduce other relevant work on this issue, and then introduce the method used in PPP. The detailed algorithms of the method are found in Appendix B.

5.2.3 Approaches to Constructivity Checking

Until now, little work has been done on the issue of constructivity except the work of Sindre [343] and Kung [220]. Since they did their works in different ways, we will take a short investigation on both.

In Kung's approach [220], the inputs and outputs to a process are treated as logical expressions with three **operators** conjunction ($\bullet$), disconjunction ($\circ$) and exclusive disconjunction ($\oplus$) showing the logical combination of the data flows. To check the decomposition of a process, a logical inference is taken with the inputs as a given condition. The rules used during the inference are of two types: **logical rules** and **non-logical rules**, i.e., each sub-process is regarded as a rule by which we infer some data flows (outputs of the sub-process) from some existing data flows (the inputs to it). For instance, from the $P_{1.1}$ in Fig. 5.6, we get a rule $P_{1.1} \models a \Rightarrow f_{1.1} \oplus f_{1.2}$. The final deducted expression can be used to compare with the output expression of the higher level process.

Sindre [343] constructs a state transition diagram on the process network which is the result of the decomposition of a process. He defines the **state vectors** making up a **state space**, and the **event space** consisting of the events of type r_f (receive an item from flow f), s_f (send an item to flow s_f), br_f (start to receive data from repeating flow f), er_f (end receiving data from repeating flow f), bs_f (start to send data to repeating flow f) and es_f (end sending data to repeating flow f). The construction starts at an initial state,

then branches are formed by choosing an event from the possible event set at the current state (the event will result in a new state). The same procedure is then repeated for all the new states until every branch has reached to the STOP state (all the termination flows have been sent out). The produced graph is used by an algorithm to create the port structure for the outputs of the network: the sequences are used to construct **and** ports, the branches are used to construct **xor** ports, and the loops are used to specify **repeat** and **conditional** flows. The constructed port structures will be simplified with some equivalence rules, and are then used to compare with the port structures of the higher level process.

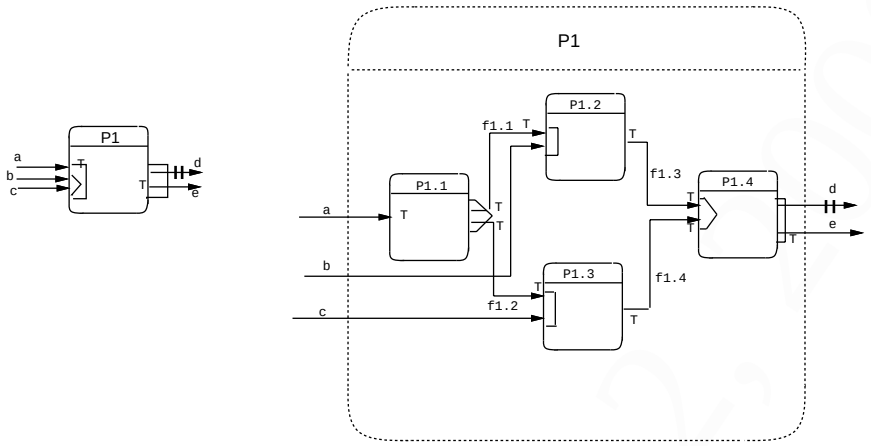


Fig. 5.6. A decomposition of a process P_1

In Fig. 5.7, we show the two ways of consistency checking on the decomposition given in Fig. 5.6.

With respect to Kung's approach, a fundamental weak point is that the pure logical method uses static means to deal with the inputs and outputs of the dynamically and concurrently executing processes. Consequently, whether a set of input flows and output flows contain data, i.e., if the logical expressions about the flows are true or false, changes over time, whereas in an ordinary logical system a true statement always keeps its truth in the whole inference process. Therefore, wrong conclusions might be drawn when we use this method. This problem has been revealed by the examples given in a paper [69].

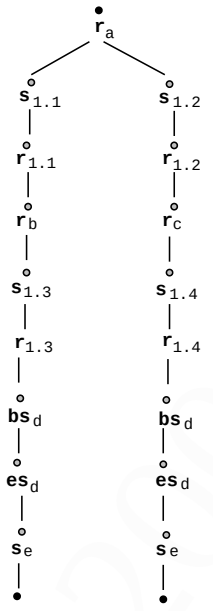
There is another big unsolved problem in both Kung's and Sindre's works: *is a decomposition itself feasible?* In other words, can the network of processes execute until some outputs are sent to the outside world? If the processes combine in an erroneous manner, then it is meaningless to talk about the correctness of the outputs from the network!

Nonlogical rules:

P1.1 $\models$ **a** $\Rightarrow$ **f1.1** $\oplus$ **f1.2**
P1.2 $\models$ **f1.1** $\bullet$ **b** $\Rightarrow$ **f1.3**
P1.3 $\models$ **f1.2** $\bullet$ **c** $\Rightarrow$ **f1.4**
P1.4 $\models$ **f1.3** $\oplus$ **f1.4** $\Rightarrow$ **d** $\bullet$ **e**

step #	results infered by the step	rules used at the step
1)	$\models$ **a** $\bullet$ **b** $\bullet$ **c**	given input
2)	$\models$ **a**	logical rules, 1)
3)	$\models$ **b**	logical rules, 1)
4)	$\models$ **c**	logical rules, 1)
5)	$\models$ **f1.1** $\oplus$ **f1.2**	logical rules, P1.1, 2)
6)	$\models$ **f1.3** $\oplus$ **f1.4**	logical rules, P1.2, P1.3, 3), 4), 5)
7)	$\models$ **d** $\bullet$ **e**	logical rules, P1.4, 6)

a) Kung's consistency check by logical inference



b) Sindre's State Transition Diagram

Fig. 5.7. Two ways for consistency checking of the decomposition of P_1

In Fig. 5.8 we give two examples to show how errors may be generated from inappropriate network structures. In Fig. 5.8.a, process P terminates when its **and** output port sends two data items into flow f_2 and f_3 simultaneously. Now process Q should be triggered. However, the **xor** input port of Q requires that one and only one input flow is full, so the output of P is actually an illegal input to Q. Figure 5.8.b gives another version of decomposition of process P_1 in Fig. 5.6. $P_{1.1}$ now has an **and** output port, so that the flow $f_{1.1}$ and $f_{1.2}$ are full simultaneously when $P_{1.1}$ terminates. It may give at least two results: if $P_{1.2}$ and $P_{1.3}$ was exactly same length of time, then $P_{1.4}$ will get an illegal input; otherwise $P_{1.4}$ will possibly be triggered two times. That is still not what we expect.

This problem is not mentioned in Kung's paper. Because errors often happen during the dynamic executions of the processes, it is difficult to use Kung's method to detect the flaws of a process model. In the other approach, Sindre had noticed the problem. His solution is the definition of a standard to determine if a network of processes is a *legal decomposition* of some higher level process:

1. *All the processes in the network satisfy the PPM syntax;*

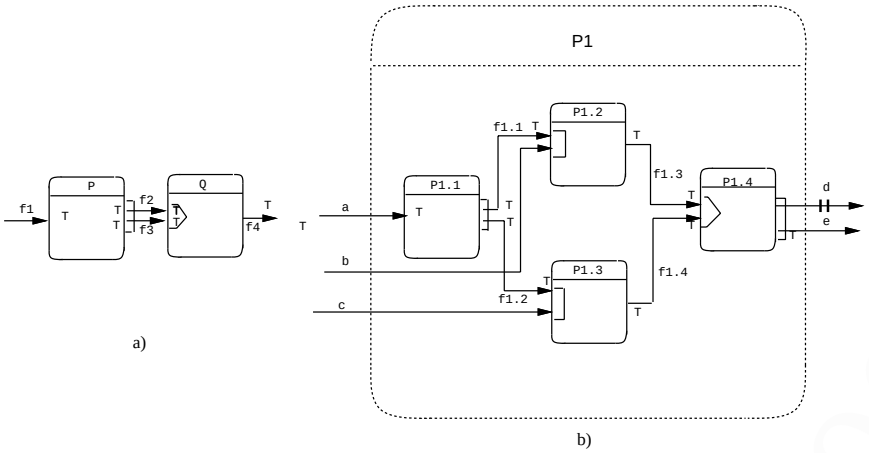


Fig. 5.8. Two process networks which may produce run-time errors

2. *There is one unique process which is triggered first in every possible execution of the network;*
3. *There is one unique process which terminates last in every possible execution of the network.*

The last two rules are used to prevent the so called "mingling" problem, i.e., the problem raised by the subsequent executions of a process network and the concurrent executions of the processes within the network – the data items in different executions might be wrongly received and processed. A similar problem is also revealed in [69].

These two rules seem to be too strong to be accepted in a practical methodology, e.g., in Tempora the rules are not followed. Moreover, even when we have imposed the constraints, mingling problem may still happen in concurrent processes between the first and last processes. Therefore, we would rather give up these two rules and leave the problem open. Here we only consider the problems of the separated executions of the higher level process when it is decomposed into a PPM.

Now we turn to the first rule given by Sindre. It states that when all the processes in a network satisfy the PPM syntax, then the network is a legal one. Just think of the examples in Fig. 5.8 where all the processes have been drawn correctly, but where errors can still happen. Thus, our conclusion is: *When we decompose a process into a network of processes, if not only each process in the network satisfy the PPM syntax, but also the interrelationships among them will not produce execution errors, then we can say that the decomposition is legal or safe.*

Such properties, as analyzed above, are difficult to check in a static way. Therefore, we choose State Transition Diagram (STD) as the basis of our

method. First of all, we define the **standard of consistency of a decomposition** as:

1. *The network can execute smoothly until the final outputs to the outside of the network have been produced;*
2. *The network receives the same inputs and produces the same outputs as the decomposed process does.*

Construction of an STD Through Canonical Process Ports. In the previous section, it has been pointed out that the more effective way of analyzing a process network is to construct an STD. However, because of the randomness of the events, even the STD for a PPM with only a few processes might be complex. To be able to construct an STD that is possible to analyze, we must find out a way for **simplifying**. The canonical port structure meets this requirement.

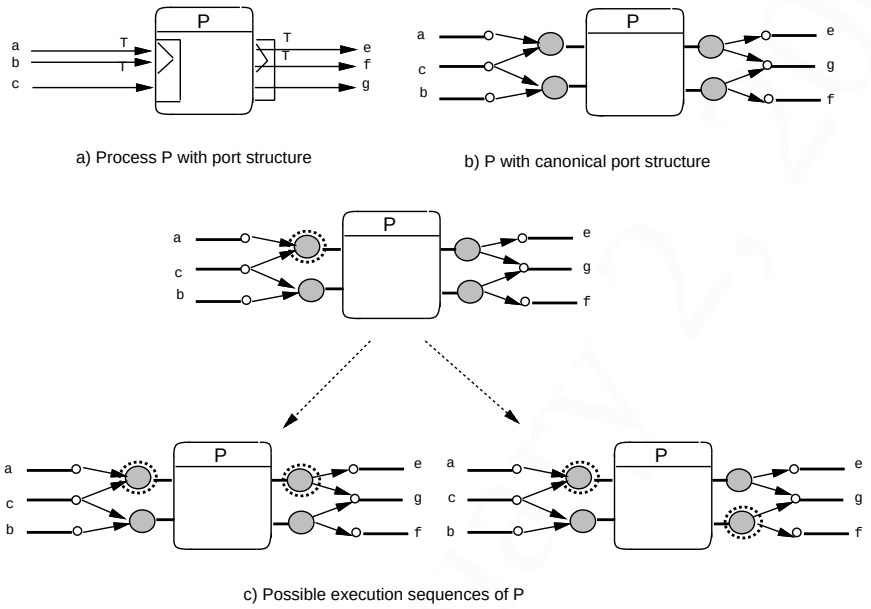


Fig. 5.9. Conducting possible execution sequences of a process

Looking at the example given in Fig. 5.9. When we consider the execution of process P given in Fig. 5.9.a, since it is a composite port, we must check if both the sub-port **xor**(a, b) and flow c are filled, then by the definition of **xor** port, we must also check if flow a or b is filled. The similar work must also be done at the output side. Moreover, if the ports are more complex, then the possible executions will be so many that the STD of it will be difficult to construct.

However, when the ports are transformed into the form in Fig. 5.9.*b*, the possible executions may be know immediately for that we know that *during any execution only one CIP and only one COP are used*. Even if the original port structures are more complex, the property of canonical ports keeps unchanged, since the structure is unique. Moreover, we know that the canonical port structure will not affect the meaning of the process since we are only interested in information about the inputs and outputs in an execution of the process.

This is shown in Fig. 5.9.*c*. Assume that any CIP enables all COPs during different executions, if the CIP **and**(a, c) is used during the execution, then we immediately know two possible executions may happen at next time, with the COP **and**(e, g) and **and**(f, g) used respectively.

Another advantage is in analyzing the behavior of the process during the process. Since CIP and COP are all **and** ports, we know that all inputs of the CIP will be received as well as that all outputs of the COP will be sent out. Therefore, the behaviors of all processes during any execution can be assumed as the sequence given in Fig. 5.10. The calculation is thus simplified both by the canonical port structure and by the assumption about the behavior of a process during an execution.

Now we define the data structure for states and then make some assumptions about the possible executions before further design the algorithms for the calculations.

The data structure for state vector

There is a lot of information associated with the dynamic executions of a process network. However, because the specification of a process network in general provide only incomplete knowledge about the process, we can only obtain limited data which reflect on the execution states of the network. We are interested in three kinds of states: 1) a flag showing whether a network as a whole is in a normal state; 2) the state of the processes in the network; 3) the states of the flows in the network. At any time, the state of the network is comprised of the states of the network component, and represented by a **state vector**. The actual data structure is given in Appendix B.

The semantics of the data structure is explained as follows:

- *normal_system_state* indicates whether the network is in a legal state;
- if the state variable *running* in a process state is true, then the process is running. Because at any execution of a process, one and only one CIP as well as one COP, is used, we will also keep information of the two active canonical ports;
- for a CIP, we need to know its ID, and how many times the event of receiving any of the inputs in the CIP has happened (for a singular input the event will happen only once, but for a repeating input the event may happen more times). Similar information should also be kept about the COP.

– For any flow, we keep a record about its volume (for a flow only linked to singular output the volume is 1, whereas for a flow linked to repeating output the volume is a integer $M > 1$), and how many data items have been put into the flow.

The assumptions on the behavior of a process network

Because we can not know the execution details of the processes, we have to make some assumptions about the behavior of a process network. Two factors are taken into account: 1) the assumptions should be as close as possible to the general behavior of process networks; 2) the assumption should depend only on the information that we can obtain from the specification of the network.

Now we list the assumptions:

1. *only the events of receiving data and sending data are considered;*
2. *during an execution of a process, one and only one CIP, as well as one and only COP, is to be used;*
3. *during an execution of a process, a singular input/output will be received or sent once, whereas the number of receiving or sending of a repeating input or output may vary. At the input side, a repeating input will be received at least once from each flow pipes linked to it and receive from every flow all the available data items. At the output side, if a repeating output depends on a singular input by the i/o condition of the process, it will be sent out twice; if it depends on a repeating input, however, it will be sent out as many data items as the repeating input receives .*
4. *the outputs of a process depend only on its inputs. This means that whenever an output condition is satisfied, it will send the output to the corresponding flow without considering whether the flow is full or not;*
5. *the inputs from outside of the boundary of the network can always be received;*
6. *a flow linked to a data store as a data resource can always provide an item to the corresponding input or receive a item from the corresponding output;*
7. **the life cycle of an execution of a process is**

begin {*triggered by arrivals of a group triggering inputs to the CIP*}
receive all the triggering inputs;
while *the process has not received all non-conditional inputs and sent out all the non-conditional and non-terminating outputs* **do**
begin
receive all the inputs that have been put into the flows linked to the inputs;
if *nothing is received (data have not arrived or all input actions have ended)*
then *send out all available non-terminating outputs;*
end {*while*}

send all the terminating outputs of the COP;
end; {*the execution*}

8. a receiving or sending action costs very little time, so during concurrent executions of the processes, we can always consider a group of receiving or sending actions of a process at a particular time point as an execution unit which is atomic relative to the concurrent executions of the processes. In Fig. 5.10, we show these "packaged events" during the execution cycle of a process.
9. whenever a flow gets more data than its capacity, or triggering inputs arrives to a running process, or two different sets of triggering inputs which can trigger two CIPs of a process are full simultaneously, the process network has fallen into an error state and can not go on with its execution.

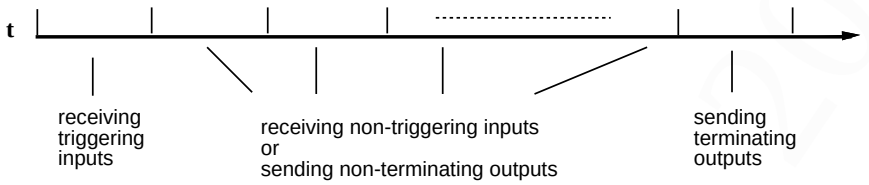
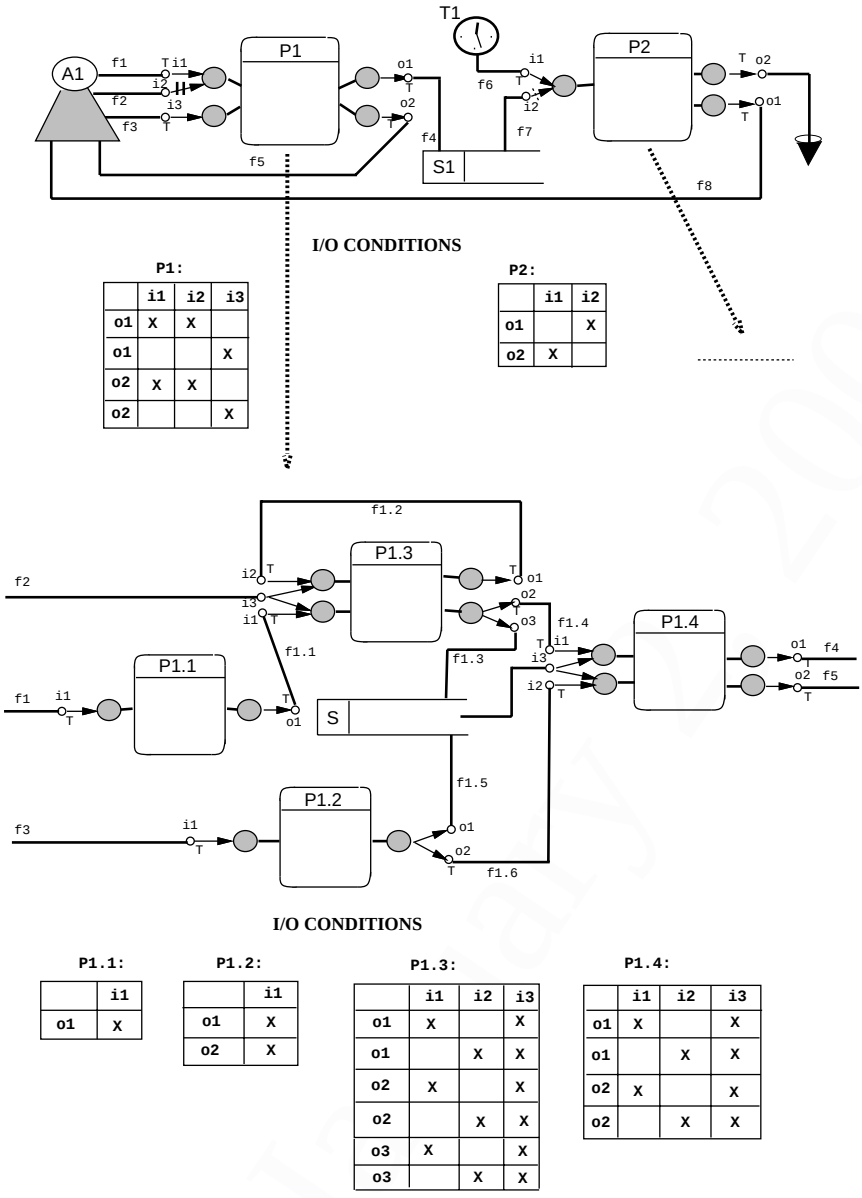


Fig. 5.10. The execution life cycle and the operation groups of a process

The algorithms to construct an STD

We use a directed graph to represent an STD: each node represents one or more state vectors and each edge is marked with one or more events that change the state of the process network at one time point to another state at the next time point. When a node represents two or more state vectors, then the possible consequent events for each of the state vectors will be exactly same. Any node represent only one state vector when it is created, but may be attached more vectors during the construction of an STD. The time interval between the states is not fixed, because we assume that each process can perform receiving or sending actions in a short time, and that it can carry out all the possible operations subsequently and independently without being interrupted by other processes. Therefore, what causes the change of one state into another is not a "single event", but a set of events occurring in a process. For detailed algorithms, see Appendix B



P1:

	i1	i2	i3
o1	X	X	
o1			X
o2	X	X	
o2			X

P2:

	i1	i2
o1		X
o2	X	

I/O CONDITIONS

P1.1:

	i1
o1	X

P1.2:

	i1
o1	X
o2	X

P1.3:

	i1	i2	i3
o1	X		X
o1		X	X
o2	X		X
o2		X	X
o3	X		X
o3		X	X

P1.4:

	i1	i2	i3
o1	X		X
o1		X	X
o2	X		X
o2		X	X

Fig. 5.11. The IFIP ticket booking activities with canonical ports

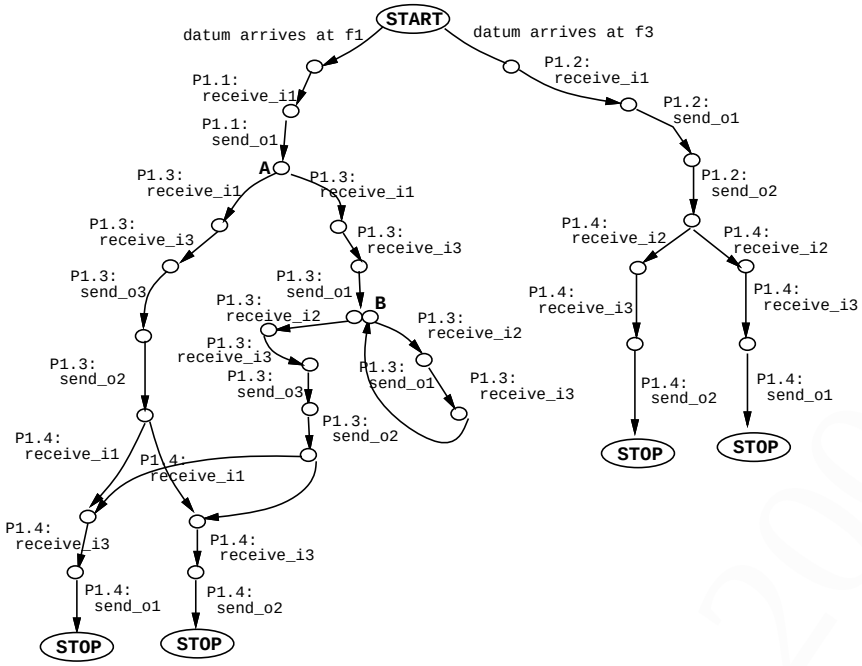


Fig. 5.12. The STD of the process network for P_1 in the IFIP ticket booking activities

In Fig. 5.12, we draw the STD for the process network in Fig. 2.45 in order to show an example. This example is redrawn in Fig. 5.11 indicating the canonical ports being used.

A small circle represents the state vector attached to a node in the graph. Usually only one state vector is put on a node, but sometimes a node may have more state vectors. In Fig. 5.12, the node **B** has two state vectors showing that the process $P_{1.3}$ (see Fig. 5.11) may be triggered with a item at its input i_1 or i_2. In the first case, the process $P_{1.1}$ just terminates and sends an item into the flow $f1.1$; in the second case, the process $P_{1.2}$ just terminates, instead, and sends an item into the flow $f_{1.2}$. Both states will result in exactly the same subsequent events, either using the canonical port for o_1 or that for o_2 and o_3, so they are attached to the same node. Node **A** and **B** also show another feature of the STD: a state can be changed into different new states by the same event (for node **A** it is the event $receive_{i_1}$). This shows that when a process is triggered in one CIP, it may have different CIP/COP pairs in its executions (also see process $P_{1.3}$ in Fig. 5.11). In the STD, every branch from the START node can reach a STOP node, so we say the process network is *feasible*.

Another example is given in Fig. 5.13, where the canonical port structure and a part of the STD of the process network in Fig. 5.8.b) are shown. The

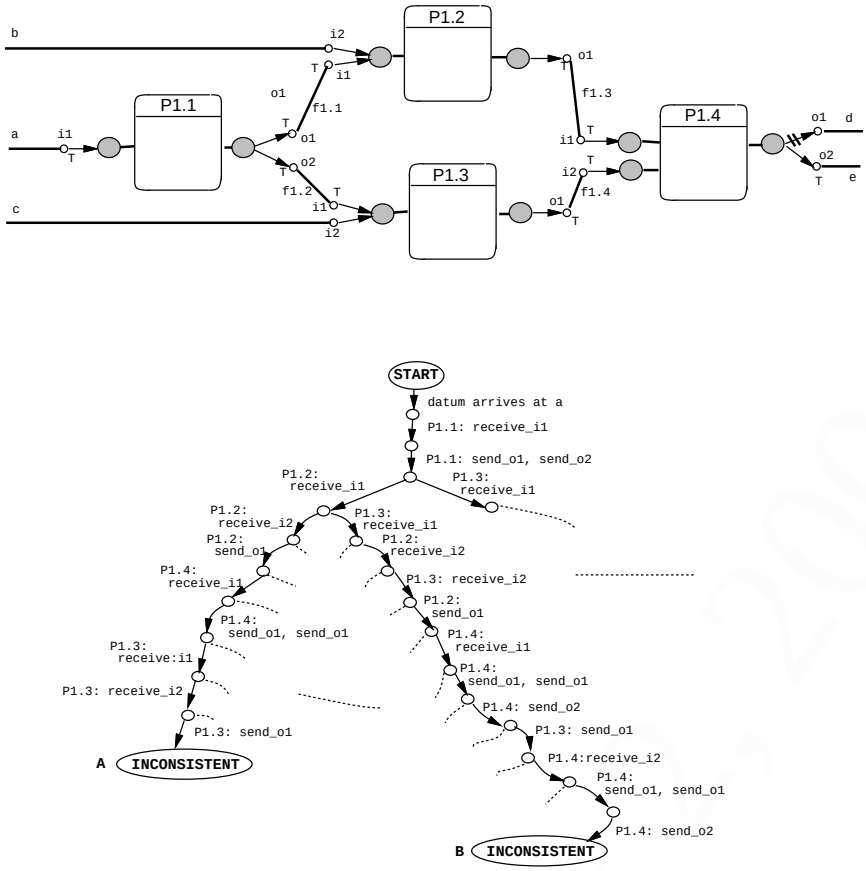


Fig. 5.13. The canonical port structure and the STD of the process network

i/o conditions are omitted since no non-terminating outputs exists in any process (thus we assume one terminating output can always be sent when the process is triggered). Node **A** is inconsistent, because when process $P_{1.4}$ is still running, process $P_{1.3}$ sends a item from its output o_1, then the flow $f_{1.4}$, which can trigger $P_{1.4}$ trough its input i_2, is full. This leads to an inconsistent state. Node **B** is also inconsistent, because process $P_{1.4}$ is triggered and sends data into the flow_pipe e twice, and the capacity of the singular flow e is only 1. In fact, all paths from node START will lead to an inconsistent state, so we say the process network is *infeasible*.

The above two examples show the extremities of executions. As a matter of fact, in analyzing the various process networks, most of them will behave differently along the different paths: some of the paths will lead to consistent STOP states and some others will lead to INCONSISTENT states. An example is given in Fig. 5.14. We say such an STD is *partially feasible*.

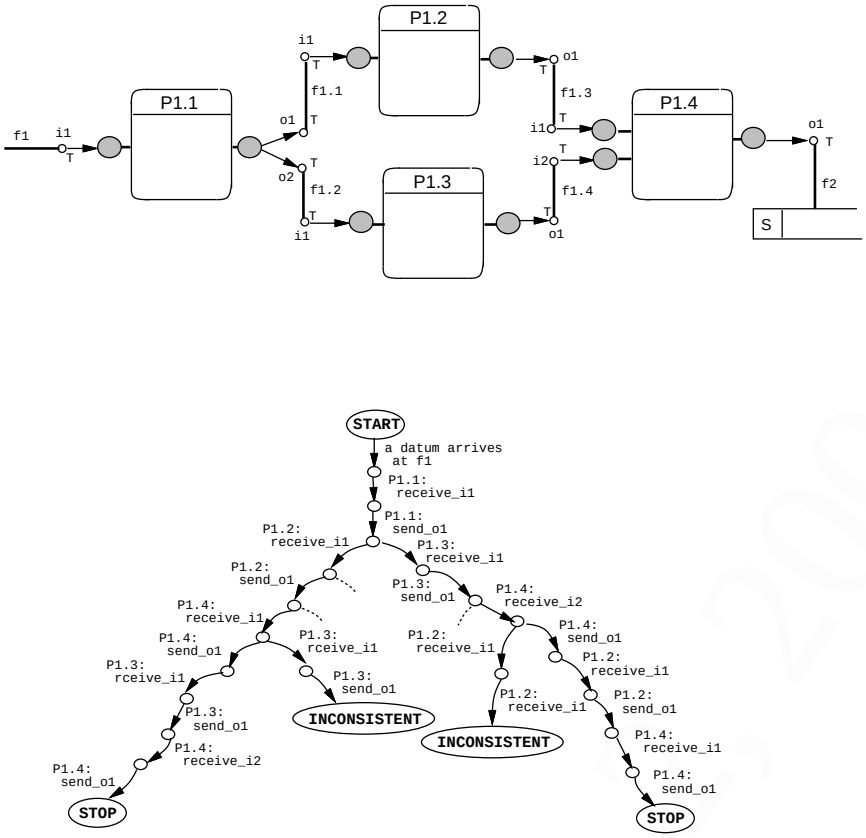


Fig. 5.14. The canonical port structure and the partially feasible STD of a process network

Because the construction of an STD is only to simulate the execution of a process network approximately and the details for control of the execution is ignored, it is possible that a process network is practically correct even though its STD is only partially feasible. However, if a network is totally infeasible, we then have enough reason to say it is not a correct structure.

When a process network is feasible or partially feasible, a further question about the network is if it has the same properties as the higher level process which is decomposed into the process network, i.e., if the network produce the same outputs as the decomposed process? Meanwhile, if the outputs to the outside of the network have the same relationships with the inputs from the outside as those specified in the i/o condition of the process?

Of the two problems, the second one has been solved by static checking. The algorithm for this was presented in the previous section.

To solve the first problem, we can construct the port structure for the network. We first find all the possible execution paths from the START state

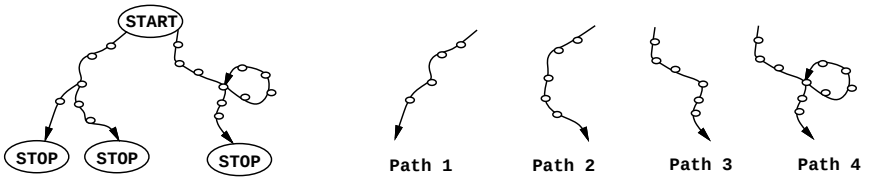


Fig. 5.15. Synthesis for the Properties of the Process Network

to a consistent STOP state. The example in Fig. 5.15 shows the principle to identify paths: each path with a particular node set is selected and identified, and the nodes within a loop should appear in a set only once except the node at which the loop begins and ends.

Each path shows an execution of the process network, thus the *receive* events at the external inputs imply an **and** input port containing these inputs. Similarly, the *send* events at the external outputs imply an **and** output port. Because in an execution the process can only follow one path, it is also implied that the **and** ports on the different paths will compose an **xor** port. However, it must be recognized that two different paths may have the events for receiving or sending data at the same or similar inputs or outputs, so the **and** ports for them should first be merged. On the other hand, the fact that some event may appear more than once or be within a cycle implies that some inputs or outputs are **repeating** and/or **conditional**. All the considerations have been represented in the algorithm Construct-Input-Port-For-Process-Network given in Appendix B.

Another algorithm, *Construct-Output-Port-For-Process-Network*, is the same as the one above except that it deals with *send* events to the external outputs rather than the *receive* events, and an output port structure is constructed for the process network.

The algorithms build the canonical port structures directly. Since the port structures for the higher level process is also transformed into the canonical form, it is convenient to compare the canonical ports to check if the process network receives and sends the same inputs and outputs as the decomposed process.

In Fig. 5.16 we construct the port structure through the STD which has eight possible execution paths in Fig. 5.12. In comparing the constructed ports with that of the original process given in Fig. 5.11 we find that the results are the same. It suggests that the decomposition is correct.

5.3 Driving Questions

A technique that can be used during modeling in a conceptual framework integrating several conceptual modeling languages such as PPP is the use

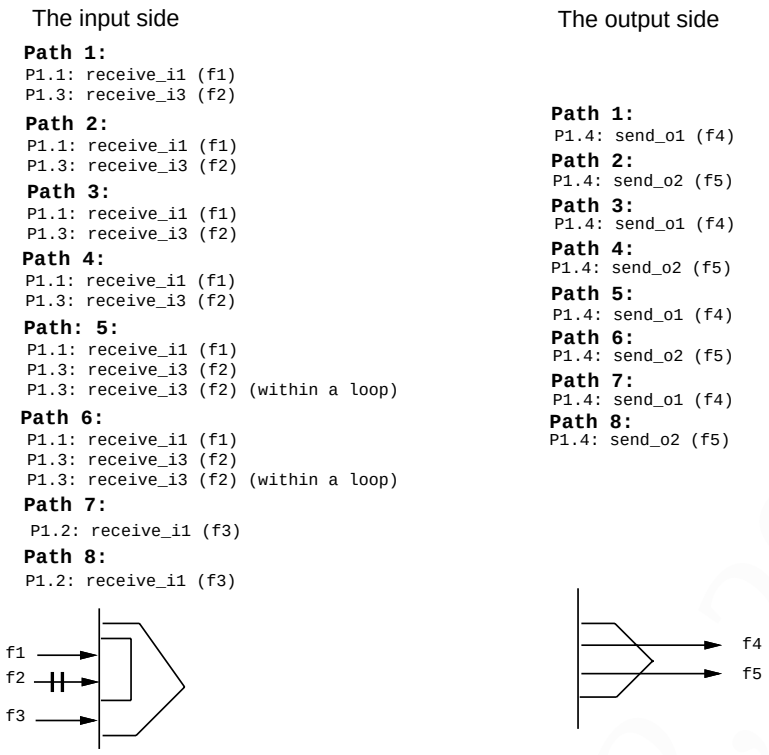


Fig. 5.16. Construction of the port structures through the STD in Figure 13

of *driving questions*. The overview here is partly based on work on Tempora [367], but extended to take the whole conceptual framework of PPP into account. Different questions will be more or less applicable according to the subset of the modeling languages used in the different modeling activities. One should also be aware of not only pursuing the avenues for modeling that open up based on the use of this kind of questions, since this can hinder the development of models that are not so easily represented in the more specialized parts of the languages.

5.3.1 ONER-Modeling

This can be performed in isolation according to guidelines for semantic data modeling. Driving questions for interaction with the other languages are given below.
Given a role in an actor-model or process-model:

– Are the structural aspects of the role represented in ONER? One might either find the role represented as an entity-class, or as the role an entity-class has in a relationship-class.

Given a process model:

- For a store: Are the contents of the store represented in an ONER-model?
- For a flow: Are the data-items traveling on this flow represented in the ONER-model?
- For a process: What data do the process use?

Given a rule:

- Are all structural components that the rule refer to found in the ONER-model?

5.3.2 Process modeling

Process modeling in isolation is similar to DFD-modeling, with the extensions indicated above. When using the speech-act modeling extension though, it should be closer to action-workflow, using conversations as the main structuring principle.

Driving questions in connection with the use of several languages are given below:
Given a rule:

- Does the rule indicate a process that is not currently found in the process model?

Given an actor or role:

- Do the actor/role have a supporting or communicating position in the process model?
- Do an existing actor/role support a process or a store that is not yet depicted?

Given an entity or scenario:

- How do instances of a class come into existence? When do they cease to exist?
- When and how are relationships established and removed?
- Is the entity being processed by any process?
- Is the entity being stored in any existing store?
- Is the entity or part of the entity-class transported as items on any flow?

Given an illocutionary act:

- Are there flows in this (or other) process models for all illocutionary acts of a given conversation?

5.3.3 Rule Modeling

The participants can be asked general questions such as if there are any specific types of restrictions that are valid for a particular area, or if there are any specific policies or types of guidelines that are used in the day to day running of the organization.
Given social actors and roles:

- What rules apply to a given role, or to a given actor (e.g. yourself) in a given situation?
- Does the rule applying to a super-role also apply to the sub-role?
- Does all rules applying to the sub-role also apply to the super-role?
- Does a rule applying to an actor also apply to any of the roles that the actor fill?
- Do all rules applying to a role, apply to all the actors filling the role?

Given a certain entity or relationship class:

- How do instances of a class come into existence? When do they cease to exist?
- For a rule valid for a subclass, is it also valid for the superclass?
- For a rule valid for a class, is it also valid for the subclasses?
- For a rule valid for a class, is it also valid for sibling classes?
- When and how are relationships established and removed?

Given a certain process:

- What is triggering a process?
- What rules applies to the internal functioning of the process?
- What are the different cases and what actions should be performed for each case? I/O-matrices may be useful here to structure the answers before they are represented as rules.

Given an illocutionary act:

- Are there any rules being implied by the performance of the illocutionary act that are not yet represented?

Given a timer:

- What is the rule resulting in the sending of each outflow?
- Are all inflows and offlows covered by rules?

Given a rule:

- Is this rule internally or externally institutionalized? If it is externally institutionalized, which internal rules need this to be in force to permit the internal rule?
- Do the rule depend on the existence of an externally institutionalized rule?
- What rules does the rule contribute to fulfilling? How else is this rule realized?

- How can this rule be realized?
- Are there alternative ways of realizing the rule than what is already indicated?
- Are there rules that work against this rule?
- Do the rule use terms which need to be defined by other rules?
- Is the rule a more restricted version of another rule?
- Are there any major exceptions for the rule?

5.3.4 Actor Modeling

Given an actor or role:

- For an organizational actor, is all its parts modeled?
- For any actor, is all relevant actors that it is a part of modeled?
- For a role, is all actors that have this role modeled?
- For a role, is all the institutionalizing actors of the roles given.
- For a role, is all relevant sub-roles of the role given?
- For a role, is this a sub-role of another role?
- For an actor, is all of the relevant roles of this actor included?
- For a support-relationship, do the supporter support in a given role? On behalf of a given actor? Is the supportee supported in a given role? On behalf of a given actor?
- For a communication, do the sender communicate in a given role? On behalf of a given actor? Is the receiver receiving this item in a given role? On behalf of another actor?
- Is an actor part of another actor in a given role? On behalf of a given actor?
- Is there a power relationship between this role/actor and another?

Given a process model:

- Is there an actor or role either communicating or supporting, that is not included in the actor-model?

Given an entity-class:

- Do the entity-class correspond to a role that is not included in the actor-model?
- Is there specializations between entity-classes that are also represented as roles, that are not indicated as super-roles?

Given a rule:

- Does the rule apply to a role or an actor that is not included in the actor-model?
- Is the rule institutionalized by an actor that is not included in the actor-model?

5.3.5 Additional Metrics for Completeness and Validity

When discussing the use of specific modeling languages, it is possible to come up with more specific metrics than could be done in Chap. 3 when discussing this in general terms.

We will here, inspired by the suggestions for metrics for requirements traceability and completeness [73] and the above come up with some proposals for metrics in connection to the use of rule-hierarchies.

- Validity: The number of rules that are not necessitated, obligated or recommended by another rule. A ratio of this compared to the total number of rules might also be interesting. One will reach a point where one is not longer able to come up with higher goals motivating for a rule [403]. These goals should be indicated specifically and not be counted when calculating such metrics.
- Completeness: The number of rules that are necessitated, obligated or recommended based on a given rule. Three cases are of special interest:
  - 0 links: The work in this area is to be completed. If the rule is a detailed rule for the processing of a process or giving explicit constraints on the ONER-model, and is thus not meant to be further specialized, the rule is not included. In the case this rule belongs to an area that it is decided to not look into further for the moment, the same applies.
  - 1 or 2 links: This might indicate that the analysis of the rule has been superficial and it needs to be worked on further.
  - The rule has a high outdegree: This may mean that the higher level rule itself is too general, and should be split into several high-level rules before being linked to the next level.
- Unresolved issues: The number of downward or-nodes, discouragement, prohibitions, and exclusions that are not addressed.

Similar metrics could be devised for the overall approach, but this is not discussed in detail here.

5.4 Chapter Summary

Semantic quality is the correspondence between the model $\mathcal{M}$ and the modeling domain $\mathcal{D}$.

The framework contains two semantic goals; validity and completeness. **Validity** means that all statements made in the model are regarded as correct and relevant to the problem. **Completeness** means that the model contains all the statements which would be correct and relevant about the domain. These correspondences can neither be established nor checked directly: to build the model, one has to go through the participants' knowledge regarding the domain, and to check the model one has to compare this with the

participants' interpretation of the externalized model. Hence, what we observe at quality control is not the actual semantic quality of the model, but a *perceived semantic quality* based on comparisons of the two imperfect interpretations. **Perceived validity** of the model externalization means that all statements that are interpreted to be part of the model is also part of the participants knowledge of the area. **Perceived completeness** of the model externalization means that there are no relevant statements known by the participant which are not interpreted to be already contained in the model.

For anything but extremely simple and highly inter-subjectively agreed domains, total validity and completeness cannot be achieved. Hence, for the semantic goals to be realistic, they have to be somewhat relaxed, by introducing the idea of *feasibility*. Attempts at reaching total validity and completeness vs. the primary domain will lead to unlimited spending of time and money on the modeling activity, thus being invalid relative to the purpose context of the modeling activity. The time to terminate a modeling activity is thus not when the model is "perfect" (which will never happen), but when it has reached a state where further modeling is regarded to be less beneficial than applying the model in its current state. With respect to this, a relaxed kind of validity and completeness has been defined.

There are many modeling activities that can be performed for establishing higher semantic quality. We have in this chapter concentrated on three:

- Consistency checking based on a logical description.
- Consistency checking based on constructivity.
- The use of driving questions to improve completeness.

6. Means for Achieving Pragmatic Quality

We have increased the font size in Fig. 6.1 of the relationships of the quality framework that are looked into in this chapter.

The main goal of pragmatic quality is comprehension. A conceptual model can be difficult to comprehend due to the formality or unfamiliarity of the modeling language used, the complexity or size of the model, or the effort needed to deduce important properties of it. A conceptual modeling environment may make use of certain *techniques* to enhance user's comprehension. Looking at the linguistic aspects of conceptual modeling, we can describe such strategies along the following four dimensions:

- **Language perception** concerns user's ability to understand the concepts of the modeling language. This was discussed in Chap. 2.
- **Content relevance** indicates the possibilities of separating between irrelevant and relevant model properties, so that at any time one is able to focus on just the relevant parts
- **Structure analysis** depends on the environment's abilities to analyze and expose structural properties of the conceptual model.
- **Behavior experience** is related to the model execution facilities offered.

A technique provides an improvement to one or several of these dimensions, thereby enhancing the comprehensibility of conceptual modeling.

6.1 Overview of Activities

Some of the activities to achieve pragmatic quality are:

Audience training. Educate the audience in the syntax and semantics of the modeling languages. We will discuss this aspect in more detail in Chap. 8.

Inspection and walkthroughs. : Manually reading a model, going through it in an orderly manner, explaining it. Useful tool-support for this in a modeling tool is support for navigation and browsing of the model. This also include the possibility of scrolling the model, either incrementally (pan) or one page at the time (page), and zooming.

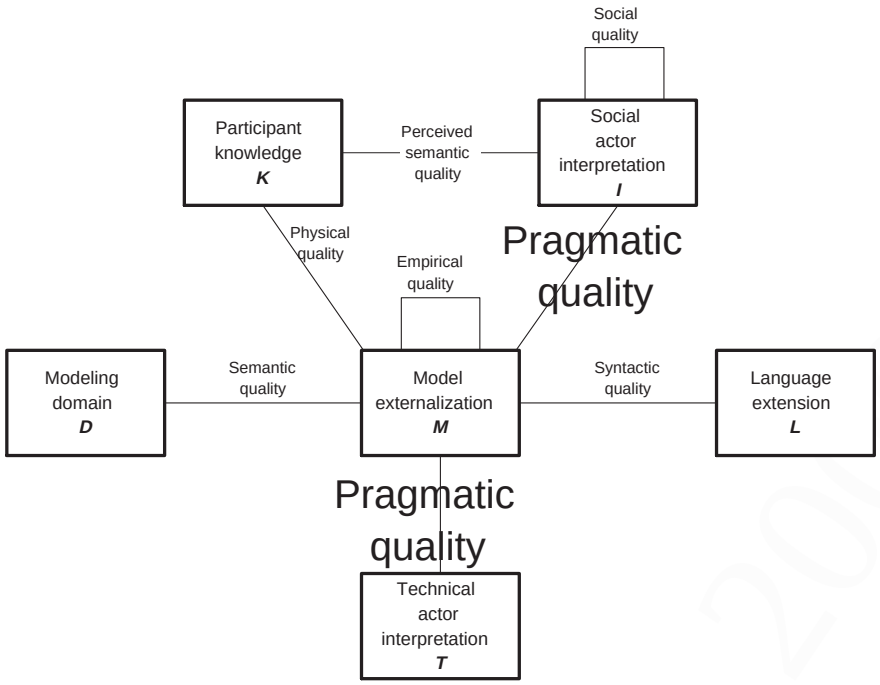


Fig. 6.1. Coverage of this chapter

Transformations. Generally to transform a model into another model in the same language. This can generally be expressed as

$$T : \mathcal{M}1_{L_i} \rightarrow \mathcal{M}2_{L_i} \tag{6.1}$$

The need for transforming models arise for several reasons. First, models may be transformed to improve the efficiency. In Gist [19] and ARIES [190], an initial operational specification gradually evolves into the final implementation by a continuous replacement of real-world modeling constructs with more efficient constructs from the programming world. Second, models or programs may have improved readability through use of transformations. This is discussed under layout modifications below. As a final example, models or programs need to changed if the underlying language grammar changes. A semi-automatic method based on the grammar changes is presented by Garlan et al. in [131]. For a set of simple grammar changes, transformation rules are derived automatically. For more complex changes, developers have to write specialized transformation procedures.

Rephrasing is a meaning preserving transformation where some of the implicit statements of the model is made explicit. One example is the use in ERAE [155] applying logical rules such as

$$\phi \rightarrow \psi \Rightarrow \neg\psi \rightarrow \neg\phi$$

. In KAOS [78], more advanced rephrasing and refinement rules are used.

Filtering is a meaning removing transformation, concentrating on and illuminating specific parts of a model. Filtering has been defined in [331] based on the notion of a *viewspec* $\mathcal{V}$, which is a model containing a subset of the statements of another model in the same language i.e. $\mathcal{V} \subseteq \mathcal{M}$.

Another way of specifying a filter is to say that it is a set of not necessarily syntactically complete deletes of statements i.e. $D_{\mathcal{M},\mathcal{S}}$.

Filters can be classified into two major groups:

– Language/meta-model filters: Suppress details with respect to graphemes and symbols in the modeling language. An example is illustrated in Fig. 6.2 where all attributes are removed.

Fig. 6.2. Example of a language filter

– Model/specification filters: Suppress details with respect to a particular model. An example is given in Fig. 6.3 where only the attributes and subclasses of a selected entity-class, in this case 'paper', are retained.

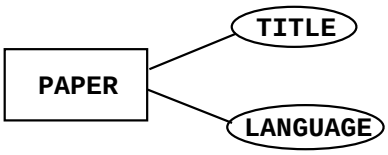


Fig. 6.3. Example of a model filter

Other relevant aspects of filters include [331]:

– Inclusiveness/exclusiveness: A filter can be defined by specifying the components to be included in the viewspec or by specifying the components to be excluded in the viewspec. This is referred to as inclusiveness and exclusiveness properties of the viewspecs, respectively.
– Determinism/non-determinism: A filter is deterministic if the resulting viewspec of performing the filter on a model M is the same each time, given that it operates on M each time. If the result is not predictable, the filter is non-deterministic.
– Global/local effects: We distinguish between two cases: (1) The scope of effects is local if there is no effect of the filter beyond the specification upon which it operates, and (2) it is global if the scope of effect is beyond

the model upon which it operates. A serious problem with filters with global effects is how to propagate changes to affected models.

Deletions and insertions are general transformations. On a higher level of granularity, these small transformations can be bundled into deltas, to differentiate between models in different versions. Making a new version of a model can thus be described as a mapping from the given model, to another model in which a set of statements have been deleted, and a set of statements have been inserted:

$$V_{\mathcal{M}} = I_{D_{\mathcal{M},\mathcal{D}},\mathcal{I}} \tag{6.2}$$

where $\mathcal{D}$ is the set if deleted statements, and $\mathcal{I}$ is the set of inserted statements.

Aspects in connection to versioning of viewspecs will be discussed in more detail in Chap. 8.

Translation. A translation can generally be described as a mapping from a model in one language to a model containing all or some of the same statements in another language:

$$T : \mathcal{M}_{L_i} \rightarrow \mathcal{M}_{L_j}, i \neq j \tag{6.3}$$

In *paraphrasing* both L_i and L_j are textual languages. Often this term is used more generally.

In *visualization*, L_i is a textual language whereas L_j is a diagrammatical language.

Translations between different diagrammatical languages can also useful for comprehension in the case different persons are fluent in different related languages. For example for those being familiar with GSM, the diagram in Fig. 6.4 might be better for comprehension than the diagram in Fig. 3.3.

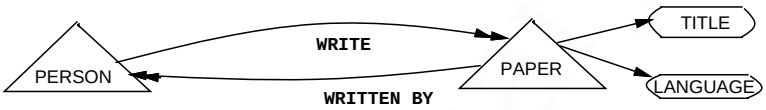


Fig. 6.4. The example written in the GSM language

Finally, one might want to translate a diagrammatical model into a textual language, for instance a programming language so that the resulting model can be executed, or natural language. In this case L_i is diagrammatical and L_j is textual.

There are a number of tasks which can be characterized as translations. By automating these tasks in modeling and CASE tools, the tedious and error-prone manual work that would otherwise be needed is eliminated. Although we are particularly concerned with translations, these are of the same nature as other syntax-directed tasks, and hence can be built using the same kinds

of support. We use the term *translation facility* to denote a system which allows translations to be specified and at least partially be implemented automatically. The need for this kind of support has long been recognized for programming environments, as seen e.g. through the development of parser generators or compiler-compilers. However, such facilities are rarely used in general modeling environments which support conceptual modeling. Rather, ad hoc procedural approaches tend to be used, where translators are dedicated to a single task and where the source and target languages are fixed.

We now briefly describe some of the syntax-directed tasks which are undertaken in CASE, with an emphasis on translations.

Multi-language Environments. Examples of work in this direction are the AMADEUS project [28], Lubars' general design representation GDR [245], and the ARIES environment [190]. Another example is given by Delugach in [88]. He uses Sowa's conceptual graphs as an internal language to translate between the ER model, dataflow diagrams, state transition diagrams, and requirements networks in SREM [5]. Of these works, only ARIES uses general translation facilities. Also, many CASE environments provide a set of integrated languages which cover different aspects of a system. It is often the case that these are semantically overlapping, so translations can be used to avoid having developers state the same data more than once. This is the case with e.g. PPP, and with PRISMA as explained in [274].

Phase Integration Support. Some environments aim at providing automated support for transitions between different phases of the systems life cycle. This can be done by translating models resulting from one phase to initial models for the next phase. Usually this can only be done semi-automatically, since the transition involves design decisions, and there is a wide range of models in the latter phase which are consistent with models from the previous phase. Hence the developers must often interact with the environment, to select an alternative if many alternatives exist. Both the DAIDA prototype [64, 187] and the IPSEN prototype [228] give support along these lines. PPP gives support e.g. in the transition from PrM modeling to PLD modeling. Also, many commercial CASE environments provide some support for phase integration.

Code Generation. Code generation can be seen as a special instance of phase integration, in that it involves translation from some intermediate model to a prototype or the final implementation. Examples of environments supporting code generation include TEMPORA [243], STATEMATE [164], PROTO [204], and PPP. Also, commercial environments like IEF use translations, for instance to port implementations to different target platforms. The majority of code generators are developed procedurally, closely tied to the source and target languages. This leads to less adaptable generators, which is not a problem as long as the environment relies on a stable set of languages. However, in more unstable situations more flexible approaches are needed.

Documentation. Documentation is tedious, but yet important work which can be partly automated by extracting and structuring data from conceptual models to form textual/graphical documents. Some environments offer the contents and structures to be explicitly specified e.g. STATEMATE, whereas other rely on a fixed specification, e.g. PPP. Many commercial environments support documentation.

Formalization. An often used method for formalizing a language, is to define its constructs through the translation to another, already formal language. Falkenberg [110] and Lubars [245] use this principle in order to define the semantics of languages for modeling of dynamic system aspects. Falkenberg uses an extended Petri-net formalism to define semantics of DFD's and action diagrams, while Lubars uses Petri-nets to define parts of the semantics of his general design representation.

Reverse Engineering and Re-engineering. Reverse engineering means finding patterns in program code or database schemas, and produce higher level abstractions from these in order to get an understanding of the underlying design and conceptual model. These abstractions are then used in the maintenance of the program, which probably was not developed using CASE technology. In a sense, reverse engineering can be seen as phase integration in the 'opposite' direction. It should be clear that when analyzing the existing code, the parsing structures of the input play a key role. Re-engineering means porting the produced abstractions into a CASE environment and thereafter go towards a new implementation, at least partly automatically.

6.1.1 Translations Facilities

Requirements to translation facilities resemble many of the requirements to software in general. Here, we will review some important requirements from the viewpoint of the translation task, and point out some particular requirements for translation.

Separate Specification Level. It has proven advantageous to have a separate specification level for translations, or at least to be able to formulate translations declaratively (Rich and Waters [313]). For this purpose, separate translation specification languages are developed. Purely procedural approaches tend to lead to less maintainable translation specifications. If the specifications are closely linked to the grammar of the source and target languages, they are more easily adapted to changes in the grammars. Note that as language grammars are specified abstractly in BNF or semantic data model form, the translation specifications should preferably refer to these rather than to the details of their repository representations.

General. The translations should be easily expressed, and the specification language should be complete for the kinds of translations needed. Furthermore, they should be adaptable, and hence easily maintainable.

Independence from Languages and Tasks. The translation specification language should be independent of source and target languages, and of the particular translation task. This way it can be used for the variety of translation tasks used in CASE environments.

Mixed Model Representations. Translations may involve mixed representations of models, in particular data model and parse tree representations. We saw that in the PPP environment, models can be stored as relations. However, expressions in PLD constructs must be parsed before models are translated. One can also imagine that such expressions are stored directly as parse trees. A particular case of mixed model representations, is when a translation is needed from a textual to a graphical representation, or vice versa. Furthermore, it should be possible to port the translation implementation to different repositories. In fact, this can be considered to be a translation task in its own.

Predefined Semantic Actions. The translation specification language should include a set of commonly used predefined *semantic actions* which are used in the production of outputs in the target language when certain patterns in the source models are found. Examples of such actions are actions to combine partial results to one overall result, to store and retrieve models, and to parse strings according to a given grammar. It should also be possible to extend the basic set of actions as necessary.

User Interaction. It is sometimes necessary to involve the developer when conflicting translations are applicable, and let her choose the most appropriate. This may for instance be the case in the transition from one phase in the systems life cycle to the next.

Automatic Implementation. Preferably, translation specifications should themselves be automatically translated to an efficient implementation. It should be possible to specify this translation using the translation specification language itself.

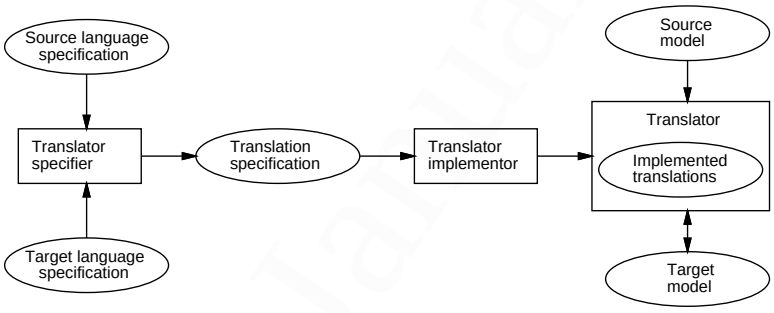


Fig. 6.5. The architecture of a general translation facility

6.1.2 General Translation Principles

For a translation between two languages, the languages need to be compatible to some extent. This means that the meaning of the source models can be preserved to some degree in the target model. If the target language can not express the same as the source model, some information will be 'lost' in the translation. On the other hand, if the target models need more information than what is found in the source models, it may be necessary to generate defaults, or to involve the developer to get the necessary data.

A *complete translation* is a translation where all statements in the source model are also contained in the target model. i.e. the mapping is an injection.

$$T : \mathcal{M}_{L_i} \rightarrow \mathcal{M}_{L_j}, \forall s \in \mathcal{M}_{L_i}, s \in \mathcal{M}_{L_j} \tag{6.4}$$

A *valid translation* is a translation in which all statements in the target model are also contained in the source model, i.e. the mapping is a surjection. For a translation that is both complete and valid, we have a bijection between the two models. It can also be useful to distinguish a completely traceable translation, where all statements in the target model is based on and can be traced back to the source model.

$$T : \mathcal{M}_{L_i} \rightarrow \mathcal{M}_{L_j}, \forall s \in \mathcal{M}_{L_j}, s \in \mathcal{M}_{L_i} \tag{6.5}$$

Another possible problem is that two-way translators may not end up with the initial model when this model is translated forth and back again. This may happen because different constructs in the source language are mapped into the same target language construct. When the reverse translation is to be done, it may be impossible to decide which original construct was used.

A general architecture of a translation facility is depicted in Fig. 6.5. First, translations are specified referring to the source and target languages. From this the translator is derived. It accepts models written in the source language, and controls the use of the implemented translations to produce a model in the target language. We refer to the two models as source and target models, respectively.

Traditionally, automated syntax-directed translation has been much studied in the context of compiler systems (Aho and Ullman [3]). Without going in details, we will present some major ideas taken from these systems. They certainly have relevance to more general translation problems. During translation in these systems, basically three steps are performed: Parsing of the input to produce a parse tree, construction of a dependency graph which determines the order in which the translation results are produced, and finally the actual production of output.

Although most translations and transformations will be easier and faster to perform when having tool support, they can also be done manually. Manual translations and transformations can also be used as part of participant training [132]. On the other hand, also audience training might be enhanced

by using tool support. Several specific applications of translations and transformations and combinations of these exist. Some examples are:

- Model execution: Translate or transform the model to a model in an executable language, e.g. the languages used for the resulting CIS, and execute this model [162, 398]. When doing this translation manually, we speak about *prototyping* in the usual sense.
- Animation: Make systems dynamics explicit by using moving pictures. This might take the form of icons such as a telephone ringing or a customer arriving at a registration desk, or it might apply the symbols of the modeling language [222, 84].
- Explanation generation: This can be manual or tool-supported. An explanation generator can answer questions about a model and its behavior [151].
- Simulation: Use statistical assumptions about the domain such as arrival rate of customers and distributions of processing times, to anticipate how a system built according to the model would behave if implemented. Neither this is practical without tool support for large models. Simulation can be combined with execution, animation, and explanation [162].

The properties a model and the languages it is made in must have include those for syntactic and semantic quality, as well as executability (i.e. the execution of the model has to be efficient), expressive economy, and aesthetics as mentioned above.

The rest of this chapter is structured as follows. A more detailed overview of techniques such as prototyping, and execution, tracing and explanation generation is given first. Then we present an overview of how these techniques including filtering have been integrated in PPP.

6.2 Prototyping

Prototyping emerged in the early eighties as an alternative/supplement to the phase-oriented way of developing systems. It is a basic component in the *spiral model* [32] and very much acknowledges the fact that users' requirements tend to change when the consequences of their requirements are presented, that is, when a concrete, tangible system is presented for the user. Subsequently, active user participation is required for a successful prototyping session [4]. In the literature, two major directions are frequently referred: *throwaway prototyping* [144, 145] and *evolutionary prototyping* [32, 71, 257, 386]. In addition to the two major prototyping approaches, there exist techniques which address more specific aspects: *mock-up prototyping*, and *experimental prototyping*. In the following, the major characteristics of these directions are explained.

Throwaway Prototyping. The main purpose of throwaway prototyping is to help the users to identify and stabilize their requirements. A main goal is to build the prototype as quickly as possible — thus, the approach is often

denoted as *rapid prototyping*. Furthermore, the prototype is experimentally used as a *learning vehicle* for both users and developers in order to gain more knowledge about the problem domain and the requirements to the future system. Thus, a prototype should focus on requirements that are poorly understood. The prototyping process is highly iterative. Several versions of the prototype must usually be made before the requirements become stable. As the name indicates, after a stable prototype has been built, it is "thrown away". The obtained requirements/domain knowledge is recorded, which was the purpose of developing the prototype in the first place.

Evolutionary Prototyping. Whereas the prototypes are discarded in the previous approach, the main intention behind evolutionary prototyping is to build prototypes which evolve into full information systems. In contrast to throwaway prototypes, the initial prototype should cover those parts of the domain knowledge that are well understood. Since users do not know all requirements prior to development, the first version of the target system can be used to play around with to gain valuable insight in the "real" environment. Also, it is expected that the remaining requirements may be clarified when the "chain" of new prototype versions evolve into a finished target system. This implies that a new version of the whole system is developed each time a new prototype is made. This way of working requires prototypes with high modifiability, and that a rather rigorous management must be employed.

Mock-up Prototyping. Mock-ups are prototypes which only reflects on the external appearance of the system (screen, reports, dialogues) with limited or no functionality [386]. Hence, no action is taken when data is input by the user, but sample data may be included to show the formats of menus, data entries, and reports. Often, mock-ups are thrown away after requirements have been defined [71]. However, screen pictures may be expanded with some functionality so that the user interface in operation can be validated. This effect is exploited in Wasserman's USE methodology [391].

Experimental Prototyping. is introduced by [119] and aims at determining the feasibility of proposed solutions. Typically, experimental prototyping takes place during the technical design of the system. It can be used to evaluate the anticipated workload of the system (performance prototyping) or to select appropriate hardware for the system (hardware prototyping).

6.2.1 A Taxonomy for Prototyping

These prototyping approaches mainly address how to rapidly create a prototype, and how long the prototype shall live during the development process. Also, techniques like mock-up prototyping states what part of the target system that should be should prototyped. Even so, the prototyping approaches are quite limited. Questions like *how much of the model should be prototyped?* or *"how exact should the prototypes represent the behavior of the model?"* are not well answered in the prototyping techniques above.

To encompass a wider variety of aspects we may classify prototyping techniques along six dimensions: *focus, scope, depth, scale, rapidness,* and *durability.* The first four aspects are introduced by [317], whereas the last two aspects are easily deduced from literature, and relates the prototype to the time schedule of development projects. The aspects may be briefly explained as follows:

1. **Focus:** i.e. what aspects (user interface, functionality, etc.) of the information system model that are of concern for the prototype.
2. **Scope:** i.e. how large a subset of the model that is represented by the prototype. There is a distinction between focus and scope [317]. Many aspects are orthogonal to its functionality. For example, the user interface of a system can be examined within a small subset of the system's functionality or across its full range.
3. **Depth**: i.e. how deeply a prototype represents the "behavior" of the model. For example, a shallow prototype of a message system might display only "canned" messages, whereas a deeper prototype might actually perform communication to obtain a more realistic prototype of the target system [317].
4. **Scale**: i.e. what volume of test data is provided for in the prototype.
5. **Rapidness**: i.e. how early in the project the prototype occurs.
6. **Durability**: i.e. how long a prototype live during the development process before it is discarded.

6.2.2 Prototyping Languages

A prototyping language is a language in which one can model system prototypes. In this section we briefly discuss *executable specification languages*, *operational specification languages*, and *very high level languages (VHLL's)*, and their relationships to prototyping and executable CML's.

Executable Specification Languages. The term 'specification' is usually taken to denote the 'what' of a system at a particular level of abstraction. If requirements are expressed in an executable specification language having user-oriented constructs, these languages correspond exactly to what we have called executable CML's. As such, conceptual models in e.g. Tempora and BNM can act as executable specifications.

However, the term executable specifications has also been used to denote specifications at lower abstraction levels, e.g. for specifications at the level of data types or subprograms. Examples include algebraic specifications (e.g. van Horebeek [375] and Goguen [140]), and specifications expressed in logic (e.g. DeVille [90]).

Operational Specification Languages. These were developed from the recognition that the what/how issues are intertwined, and that the division made

should rather be on problem-oriented versus implementation-oriented concerns. Two examples are the PAISLey [408, 409] and Gist [19, 20] languages. As operational specifications are closely connected to a particular development approach, we prefer to use the term 'executable CML'.

Very High Level Languages. These are most often general purpose programming languages. The "very high level" indicates that complex data types and operations are built into the languages. This results in more compact, more declarative, and less efficient programs than those written in a traditional HLL. There are two ways to exploit these languages for prototyping. One is to use them in a loosely coupled approach. Another alternative is to use these languages as target languages when prototypes are generated from an executable CML. As the constructs offered by VHLL's are usually more implementation-oriented than those found in CML's, they are not that well suited for conceptual modeling.

Example languages include 4GL's and database programming languages, PROLOG, and functional languages like MeToo [167]. 4GL's are tailored to development of database applications, and give opportunities to specify tables, reports, screens etc. PROLOG provide lists as the primary data type, but arbitrary complex terms may be used. Its clause resolutions provide a powerful inference mechanism. Tavendale [365] and Weigand [392] both use PROLOG for prototyping from conceptual models.

6.3 Execution of Conceptual Models

If the conceptual model is directly interpreted, the model is the prototype, and no translations are necessary. If the model is translated to another executable language, the differences in abstraction level and styles of expression will affect the ease with which the translations are specified and implemented. Here, we discuss three different categories of target languages: Traditional high level programming languages (HLL's), very high level programming languages (VHLL's), and executable CML's.

Execution by Direct Interpretation. Developing an interpreter is similar to specifying a language's operational semantics. To each construct of the language, a *semantic function* is specified, which gives the computation to be performed when that construct is recognized in the model. Consider the simple example a:=x Op y, where a is a simple variable, and x and y may be complex expressions. When such a statement is recognized at run-time, it is evaluated by first evaluating the values of x and y, and then the operation Op is performed on the resulting values. Finally, the value returned by evaluating the operation on the arguments x and y is stored for the variable a.

Although this example may indicate that developing interpreters is a simple task, it is not, for all but the simplest languages. CML's may be rather complex, both in the data structures and control structures offered. It may

be a challenge to develop an interpreter with an acceptable performance. As an example, the interpreter developed for the Gist language was too slow to be used for practical prototyping, so translations to a more efficient language were introduced to get sufficient performance (Feather [112]).

However, when an interpreter has been developed, and the CML is relatively stable, certain advantages are offered. Naturally, translations are not necessary after each model revision, and hence the feedback loop is shorter. Changes to models can be made in the editors, and tested on the fly. Also, by controlling the execution through interpretation, it may be easier to link the model execution to tools which provide supporting validation techniques like e.g. explanation generation.

Examples of interpreted languages include Transformation Schemas in Teamwork [30, 390], Statecharts in STATEMATE [161, 164], SXL [227], PAISLey [408], and structured analysis languages presented by Lea and Chung in [225].

Execution by Translation to HLL's. Although some programming languages have constructs for concurrent computations, for the most both data structures and control structures are relatively simple. They generally have a procedural rather than a declarative style of expression, and are characterized by high performance. As HLL's usually are the ultimate implementation languages for systems, it would be attractive to generate as much as possible of the systems directly from their conceptual models. However, as the difference in abstraction levels and in styles of expression (procedural vs. more declarative) is quite large, translating a conceptual model into a program expressed in a HLL is not easy. As an example, consider the problem of simulating a database with instance manipulation and checking of constraints in a HLL.

Still, for certain aspects of a conceptual model, or for relatively simple CML's, translations to HLL's are feasible. One example is REMORA [241] where prototypes are generated in PASCAL with embedded SQL.

Execution by Translation to VHLL's. As we described above, VHLL's may be suitable target languages for prototype generation from conceptual models. Besides having built in more complex data types, these languages tend to have a more declarative style than HLL's, e.g PROLOG and 4GL's. This is likely to make translation specification easier, but the cost is likely to be decreased performance. As these languages are primarily programming languages, their constructs are not that of CML's.

Examples of environments using VHLL's include ARIES [190](a database programming language called AP5), and REMORA [241](SQL) as explained above.

Execution by Translation to Executable CML's. If another executable CML is used as a target language, it is assumed that a sufficiently efficient executor exist or can be developed for this language. Translations are likely to be easier to specify if the target language is expressive, and provides the dominating perspective(s) of the source language. As with VHLL's, the price is

usually slower computations. The target language may again be interpreted, or translated to yet another language for efficiency reasons.

If the executor of the target language has sufficient performance, the potential ease of translation specification and implementation gives a great advantage for multi-language environments, and for meta-CASE environments. Also compared to the task of developing interpreters for each new language, translations to an executable CML is easier.

We see examples of translations to executable CML's in multi-language environments like ARIES [190], where external languages are translated into the internal representation language, and with Lubars' general design representation (GDR) [245].

6.3.1 Execution Mechanisms

An executable model is in many respects similar to a program. It can be interpreted or compiled, it receives certain inputs and produces certain outputs during execution. The similarities induce a need for mechanisms comparable to those found in programming environments (Harel et al. [164]), particularly as those found in program debuggers. By supporting features like *step-by-step interactive execution*, *(programmed) batch execution*, *breakpoints*, *spypoints* etc., the developers and users achieve better control of the execution.

Step-by-step interactive execution lets the users or developers respond on behalf of the system environment, giving inputs as external events to the executing model. The model responds to the events according to its specified dynamics in a single step, which updates the model state. Further steps may be initiated until the model reaches some equilibrium. After each step the user can inspect the model state, and possibly report new events. Execution in this fashion is found in e.g. Teamwork [30] and STATEMATE [164].

The alternative to this interactive mode of operation, is to store or program all events on a separate file, and then run the models as batch jobs. This naturally limits users involvement to observation of outputs, but it may be useful if complex, variating scenarios of system use is to be set up. Programmed execution allows inputs to be represented as statistical distributions, as done when executing models written in simulation languages. In STATEMATE [164], *exhaustive execution* is proposed as a means to test critical components by generating all possible sequences of inputs, and check the model for unwanted properties like deadlocks or unreachable states.

Further control can be gained by inserting breakpoints or spypoints in the executable model, similar to their use in a program debugger. A breakpoint is a statement, which, when reached, stops the execution to let the system state be inspected, state components be updated etc. Spypoints are used to record events for later inspection, or to report to users or other tools about occurring events.

6.3.2 Requirements to Tools Supporting Executable Conceptual Modeling Languages

Here, we are only concerned with the support for executable CML's. We divide requirements into three categories. 1)Requirements to the CML's used, 2) requirements to the model executor, and 3)requirements for facilities which help in understanding the model execution.

Language Requirements. The CML's expressiveness should be such that any particularly uncertain aspects of the modeled systems can be investigated through model execution. Naturally, the languages should be executable, but we also require that they should not lead to intolerably slow prototypes. A significantly lower performance than for the final system is expected, though cf. Chap. 2.

Executor Requirements. The basic requirement here is that executors should take conceptual models as input, and either interpret them directly, or make use of a translator to map them into an executable representation which is then executed. During execution, inputs required and results produced should be presented in ways comprehensible to the users, although a realistic user interface is not expected from a prototype focusing purely on functional aspects of the modeled system. Another requirement concerns the integration with the modeling editors. To ensure a short feedback loop, it should be easy to make changes to the models and subsequently start executions again to observe the effects of model changes. Further, it should be possible to control the execution in various ways, for instance by providing debugging aids like step-by-step executions, breakpoints etc. Finally, during modeling, temporary errors like incompleteness and inconsistencies should not necessarily make model execution impossible. Rather, defaults could be added, or the users could be asked for necessary information during execution, or only parts of an overall model could be selected for execution.

Requirements for Understanding Executions. There are various supporting techniques which can be used to enhance the understanding of the observed external behavior of an executing model. Ideally, the environment should offer a battery of techniques like model animation, tracing and explanation generation in order to cope with different execution situations. We will look upon this in more detail below.

6.4 Tracing of Model Execution

From the literature, we find that execution traces are used for various purposes.

Program Debugging. When used for program debugging, execution traces of programs can be inspected to search for errors made. This is especially needed for concurrent programs which communicate with each other, and together form very complex behavior (McDowell et al. [259]). More advanced use of traces than mere inspection, are program visualization and replay of executions. Replaying of executions are necessary because of the possibly nondeterministic behavior of time-critical concurrent systems. The traces may represent e.g. updates of variables, interprocess communications, procedure calls etc. When an error is detected, the execution trace is compared with the program to identify the cause(s) of the error. This search is normally manual, guided by some hypothesis made beforehand. Some tracing components provide effective means for browsing through traces, or for retrieving traces much like database queries.

Paraphrasing System Dynamics. Execution traces tend to be very large and complex, and the search for useful information is difficult. One way of describing the model dynamics, is to execute the model, record the occurring events in a trace, and then paraphrasing interesting parts of the execution trace. In Gist [359], symbolic execution traces are paraphrased by a 'behavioral explainer'. It analyzes traces to find interesting fragments, and builds up an explanation structure which is subsequently paraphrased into English text. The main problems are to select relevant data from the traces, and also to paraphrase proofs. The former problem is solved by applying a set of heuristics, each which recognize a situation which is deemed useful for the developers or users to know about. The latter problem is approached by searching for instances of well-known inference rules like modus ponens etc. There are no possibilities for users to ask specific questions, rather, they must rely on the heuristics put into the explainer.

Another example of paraphrasing of execution traces is presented by Kalita in [198]. There, natural language status reports are generated from a set of inter-related processes described by Petri-nets. This system is similar to the explainer in Gist, but it is based on concrete rather than symbolic executions.

Validating Behavior. Benner [24] describes the use of execution traces in the ARIES simulation component. He uses *validation questions*, which are patterns of behavior, to search for desired or undesired behavior from the trace. The patterns are sequences of events to be searched for in the temporal order they are given. The patterns contain variables which are instantiated when the patterns are matched with the trace being produced. A fully instantiated pattern means that the specified behavior is observed, be it a desired or undesired behavior. If a desired behavior is only partially matched, or not matched at all, this means that the model has errors.

Explanation Generation. Explanation generators may be used both in the development phases for model validations, and in the final system in order to help the end-users. Most research on explanation generation rele-

vant to our work has taken place in expert systems research, e.g. Basu et al. [22],Chandrasekaran [59], Clancey [66], and Neches et al. [272]. As we will briefly describe later on, these explanation generators are able to explain more than execution traces. However, we will focus on their ability to recognize and answer a range of questions related to the behavior of executing models below.

6.4.1 Requirements to Tracing Components

In this section we sum up some major requirements to a general tracing components. Many of existing explanation generators do not have separate tracing components, rather some intertwine the problem solving status with their knowledge structures e.g. in RATIONALE [1]. However, these traces tend to be rather simple compared to traces from executing conceptual models.

Language Independence A general tracing component should be adaptable to different modeling languages and model executors. This can be achieved by either basing the execution trace on a single, but very general schema, or by allowing different schemas to be specified for each application. Note that schema here is used in a general sense, it simply gives an intentional description of execution traces.

Expressiveness Naturally, all data deemed necessary for later retrieval should be traceable. The complexity and detail level of executable conceptual models is comparable to that of programs. Hence, we should study the tracing techniques used in programming environments. The data stored in traces includes state changes, the reasons for state changes, and dependencies and relationships between state changes. The latter includes temporal and hierarchical relationships.

Event Reporting It should be easy to report about relevant events to the tracing component, without disturbing the behavior of the executing model. Also, it is important that the temporal ordering of events is maintained in the trace.

Data Retrieval To make use of the stored execution trace, it must be possible to search for information in an effective way. For large and complex execution traces, there should be facilities going beyond mere inspection. As we noted above, different browsers and query facilities have been developed to study program traces. For explanation generation, particular queries should be made available, so that data to include in explanations can be easily retrieved. The form of queries supported should be based on the questions users may ask about model behavior, and it should be possible for a developer to use queries directly to understand executions based on traces.

Non-functional requirements These concern the performance of the tracing component, including efficiency and reliability.

6.4.2 General Tracing Principles

The architecture depicted in Fig. 6.6 represents what may be called a database approach to tracing. It is more close to the way execution tracing is performed in program tracers than to the way it is performed in most current expert systems.

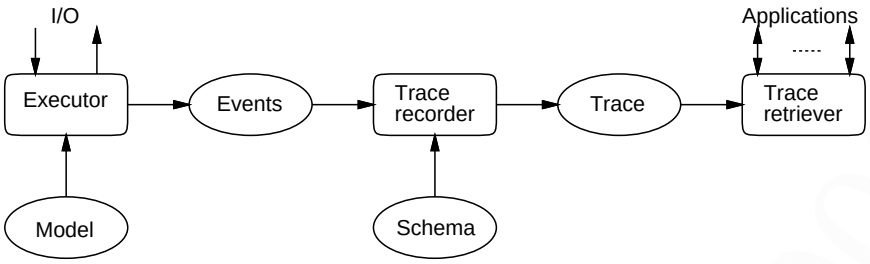


Fig. 6.6. A general architecture of a tracing component

In the architecture, a model executor reports about occurring events as necessary, for instance by calling specialized routines for each event category. At the program level, events typically correspond to memory accesses, message interchange, object access, procedure calls, tasking activities etc. In an expert system, events may correspond to e.g application of rules, conclusions reached, and hypotheses refined. In an executing conceptual model, events reported may be about data flows sent, processes or activities applied, updates of state components etc. Usually, a fixed set of events is defined, but some systems provide *event definition languages* so that they can be more flexibly specified. This tend to create a considerable overhead, because events need to be detected at runtime.

Also, the detail level and the amount of data to be stored depends on the future use of the trace. For instance, to replay executions in non-deterministic situations, it is enough to store data which will resolve the non-determinism. However, in order to explain execution behavior, it is expected that very detailed data about the executions must be stored.

In any circumstance, the data to be stored must somehow be reported. Usually, this is done by inserting *probes* into the executing model. The probes are calls to routines which transfer data to a *trace recorder*, or they store data directly into the trace. To do this, they must be placed at appropriate locations in the model. If the models are translated to prototypes automatically, it is preferable to generate the probes as well. If the models are directly interpreted, then the probes may be integrated with the interpreter. In some cases, the execution of probes may change the behavior of the executing program or model. This may particularly be so in time-critical, concurrent systems, as noted by McDowell [259]. We will not study this problem here.

Continuing towards the right of the architecture, the reported events are stored in the order they occur by the trace recorder. The data is stored according to a predefined schema, which includes attributes for temporal ordering of events. Many systems use a global clock in order to achieve the ordering, and mark starts and ends of intervals. This clock may represent the actual time, or provide a state numbering, e.g. each time an event is reported, the state number increases.

Finally, in the database approach, queries are used to specify the data to be retrieved from the trace. For explanation generation, these queries form the interface to the execution trace. Also other tools could use the execution trace through the interface provided with trace queries, and if they are made user-friendly, developers can use trace queries directly.

6.5 Explanations Generation

As a provisional definition, let us assume an explanation to be *a text, or a combination of text and graphics, that describes a phenomenon and satisfies some user's need for information.* Although vague and superficial, this definition suffices for the presentation to follow here. We make the general assumption that both the content and the structure of an explanation should vary according to the properties of the user requesting the explanation, the reasons for requesting it, and the context in which it is requested.

What this indicates, is the explanations' dependence on general text linguistic theories about how humans communicate with each other. Let us then take a typical conceptual modeling process. We can distinguish between two process activities, the *construction* and the *validation* of models, assuming both to be able to take advantage of explanation generation technologies. In the modeling, there are people involved that bring with them different roles, skills, and attitudes.

In the following, we will demonstrate the usefulness of explanations with reference to the construction and validation of conceptual models, using illustrations from a bank system.

6.5.1 Construction of Models

Ideally, the modelers have intimate knowledge of both the syntax and the semantics of the conceptual modeling language. They feel comfortable with the language, and find it easy and effective to use in the modeling process. In practice, though, one cannot always assume this to be the case:

– Formal languages are notoriously detailed and complex. The emphasis on compact, unambiguous and effective ways of representing information reflects a gap of philosophy between these languages and natural languages that is not easily bridged.

- Since a single modeling language is usually attributed to only one perspective of a system, one will often have to switch between languages, or use several in combination. This complicates the learning of the languages, since one do not get the time to develop a proper understanding of them.
- To an increasing extent, domain experts or end-users are getting involved in the modeling process. These cannot always be expected to know and be familiar with the conceptual modeling language used in the project.
- The complexities and uncertainties of the domain itself make it difficult to concentrate on the details of the modeling language. Since it is more important to investigate the domain than to construct a syntactically correct model in the beginning, complicated features of the language tend to be neglected if they disturb the analysis of the domain.

In search of improvements in conceptual modeling, a wide range of works has been done withing the fields of modeling languages and tool support. Abstraction mechanisms, user-oriented concepts and graphical symbols are now generally recognized as important to a language's comprehensibility, but the effect of these strategies is hampered by the overall need for formal and expressive languages. And even though tool support like syntax-oriented editors, model checking routines, and help facilities is now often included in modeling environments, there are still problems with the comprehensibility of the modeling process.

Explanations can help the modelers with the language's syntax and semantics, and if a verification module is included in the environment, they can also provide an interface to error messages from verification checks. Verification rules may be classified as *a priori* or *a posteriori* [379]. An a priori rule can never be violated (e.g. A flow in a PrM model can never go from one store directly to another), whereas an a posteriori rule can be temporarily violated in the modeling process as long as the final model obeys the rule (e.g. all processes in a PrM model must have at least one output flow).

In connection with the construction of conceptual models, there are at least two important explanation contexts, the *meaning* context and the *reference* context. In the meaning context, the focus is on how terms refer to underlying meanings, in the reference context on how terms refer to constructs of the modeling language.

Language semantics. The modelers can ask questions concerning the meaning of language concepts. Since the semantics of conceptual modeling languages seldom are explicitly represented, we will probably have to suggest — rather than to precisely specify — concept meanings. It is possible, however, to indicate semantic properties by describing for example the concept's type, purpose and/or structural relationships to other concepts. Consider the dialogue below:

user: *"What is a process?"*
system: *"A process is a dynamic concept that generates a number of outflows on the basis of a number of inflows. Its content is described as a separate diagram or as a set of process logic rules. It is triggered by the arrival of a flow."*

The first sentence tells us about the type and purpose of processes, whereas the second one says how they are related to other language concepts. In this context, the question was assumed to refer to the underlying meaning of processes (i.e. the meaning context was used). Looking at a conceptual model a user might also wonder what graphical symbols are referring to processes. She is then in the *reference* context, and the question *"What is a process?"* could then lead to the graphical explanation in Fig. 6.7. Note that explanations with graphical elements involve the generation of *views* of the conceptual model.

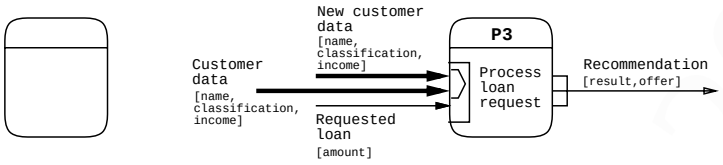


Fig. 6.7. An explanation showing what a process looks like

Language syntax. Questions concerning syntactic properties of the modeling language are all in the reference context. They request information about how constructs of the language are related to other constructs, and can draw on a priori rules as well as a posteriori rules. The question below refers to the processes used in PrM .

user: *"How are processes used?"*
system: *"A process has an identifier and a purpose. It must receive at least one data flow and send at least one data flow. All the inflows must be included in an input port construction, all the outflows in an output port construction. At least one inflow must be marked as triggering. A process's content is described as a new diagram or as a set of process logic rules."*

If the system has reasons to believe that the modeler is familiar with the traditional data flow diagrams, it can leave out certain structural properties that are already known to her:

system: *"All inflows to a process must be included in an input port constructions, all outflows in an output port construction. At least one inflow must be marked as triggering. A process's content is described as a new diagram or as a set of process logic rules."*

In this way, the response is tailored to the user's knowledge. If the user has no experiences with process-oriented languages, the original explanation accompanied by a model example like the one of P3 in Fig. 6.7 could be appropriate.

Verification checks. Most modeling environments include syntax-oriented editors, in which only constructs from the alphabet of the language are available in the modeling process. The environment will usually also offer some routines for checking the syntax, the consistency, and internal completeness of the model being built. If these verification routines are run on the model in Fig. 6.8(a), for example, the lack of flows out from the process is detected, and the explanation below might be generated:

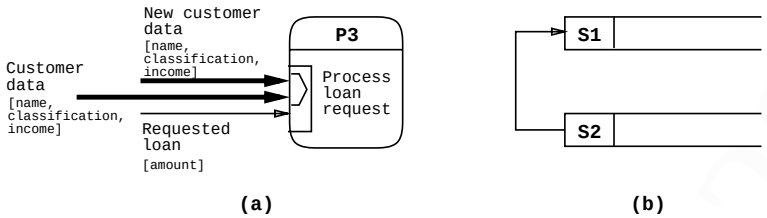


Fig. 6.8. Illegal constructions in PrM . In (a) an a posteriori rule is violated, in (b) an a priori rule

system: *"The diagram is illegal, since* P3 *does not produce any flow. A process must produce at least one flow."*

The rule used above is an a posteriori verification rule. If a flow connects two data stores — as is the case in Fig. 6.8(b) — an a priori rule is violated and we can get a message like the one below:

system: *"The origin and the destination of a data flow cannot both be data stores."*

Note that a good editor would not allow the user to violate a priori verification rules. In that case, an explanation component could be used to inform the modeler why her construction was not accepted.

6.5.2 Validation of Models

The product of modeling is a conceptual model that serves as a basis for the design of the information system. Since conceptual modeling is an iterative process of model building and model analysis, though, there will be many partial and temporary products before a final, satisfactory conceptual model has evolved. An important task in the analysis of these products is to

determine their appropriateness with respect to user's needs and intended requirements, i.e. model validation. In order to do that, the parties involved in the modeling must be able to understand the properties and the consequences of those parts of the models that are relevant to them.

Apart from the formality and unfamiliarity of the modeling language, though, these models may be hard to validate due to their *scope* and *complexity*.

- The scope of the conceptual model concerns both the number of phenomena modeled and to what level of detail they are described. In many cases, the model includes much more information than what is relevant to a particular user reading the model. The large scope of the model can disturb the user's ability to concentrate on and validate those parts that she is knowledgeable in.
- A conceptual model of an information system easily gets very large and complex. There are perceptional problems in reading very intricated models, and the details of these models tend to overshadow their overall structures and behavior.

So, in realistic information systems development projects, even models represented in familiar and not so formal languages can be hard to interpret due to the inherent complexities of information systems.

Strategies for enhancing user comprehension. The previously mentioned strategies for improving the comprehensibility of modeling languages have also profound effects on the validation of the corresponding models. In addition to these, there are other strategies that expose model properties through heavy analyses or model manipulations, like *view generation, model execution*, and *model translation*. These strategies are all valuable, though they are restricted to specific aspects and have certain limitations.

What validation is concerned, explanations can help the users read the conceptual models correctly. If modeling decisions are recorded, they can also justify the inclusion of elements in the model. At last, they can guide and explain executions of these models, and in that sense they provide an interface between executable model or trace, and user.

Model inspection. The users can ask questions about properties of the conceptual model. The questions may be related to structural or dynamic properties, and the answers are user-tailored and context-dependent. An explanation can be adapted to the user and the context, as indicated by the three alternative explanations below:

user: *How is* P2 Register customer *activated?"*

system$_1$: *"*P2 *is activated when either* Customer_name *is sent from the user or* No_registration *is sent from* P1. *The user sends* Customer_name *if she wants to register a new customer.* P1 *checks the customer and is triggered by a* Loan_request *from the user.* No_registration *is sent if the customer is not found in the database and* No_customer *is returned to* P1.*"*

system$_2$: *"*P2 *is activated when either* Customer_name *is sent from the user or* No_registration *is sent from* P1.*"*

system$_3$: *"*P2 *is activated if the user wants to register a new customer or a customer not already registered requests a loan."*

The first explanation could be suitable for a modeler inspecting the details of the model. The next explanation could be used in the same *inspection* context, but in this case either the user knows the details left out from the first one or the context makes the origin of Customer_name and No_registration irrelevant. In the third one, we have changed to a context where user's interaction with the system is emphasized. The explanation could be given to a user unfamiliar with the details of the system, but with a good understanding of how the system is expected to be used.

The next dialogue illustrates how different levels of abstraction can be exploited to tailor a particular explanation:

user: *"How is* Loan_limit *computed?"*

system$_1$: *"*Loan_limit *is produced by* P3.1. *If* New_customer data *is received, then* Loan_limit.maximum = New_customer_data.income. Loan_limit.name *is set to* New_customer_data.name. *If* Customer_data *is received, there are three cases:*
If Customer_data.classification *is* 'bad', *then* Loan_limit.maximum *is 0.*
If Customer_data.classification *is* 'neutral', *then* Loan_limit.maximum *is* Total_savings.savings + Customer_data.income - Total_loans.loans.
If Customer_data.classification *is* 'good', *then* Loan_limit.maximum *is* Total_savings.savings + Customer_data.income - Total_loans.loans.
Loan_limit.name *is set to* Customer_data.name.*"*

system$_2$: *"*Loan_limit *is computed on the basis of* Total_savings, Total_loans, *and either* New_customer_data *or* Customer_data.*"*

The first explanation describes in detail how Loan_limit is computed, and would be appropriate when the model is validated by users very knowledgeable in the domain. The second one gives a functional overview of the computation. It is assumedly most helpful in situations where the overall structure of the model is in focus, or the user cannot be expected to be able to get anything out of all the details or be able to comment meaningfully on the basis of them.

The following dialogue is an example of a request about structural properties:

user: *"What is the content of flow* Loan_request*?"*
system: *"*Loan_request *contains two items,* name *and* amount*."*

Of course, the content of these explanations could also differ according to explanation context and user characteristics.

Modeling deliberations. Modeling deliberations typically express the arguments for including elements in the conceptual model. They can be made a part of the conceptual model, and can be used by the explanation generation component to justify the content of the model.

Consider this typical question about modeling deliberations:

user: *"Why is* P2 *needed?"*
system: *"*P2 *is needed because the bank must be able to accept new customers."*

Depending on the sophistication of the recorded deliberations, we can justify the inclusion of elements (as above), explain why elements are considered a correct representation of some phenomenon, or in general provide arguments that support or refute the existence or use of model elements. Provided that more detailed deliberations are included in the conceptual model, the following dialogue could be possible:

user: *"Why is* P2 *correct?"*
system: *"To register a new customer, only her name and income is necessary. Since the bank has not served the customer before, no classification is available, and it seems reasonable to assume a* 'neutral' *classification to start with."*

Modeling deliberations are seldom formalized, so the explanation component's task would normally be to track the appropriate arguments and just make them available to the user. In the conceptual models, deliberations are modeled as special kinds of model elements and are linked to the other elements of the model. We will from now on not distinguish between conceptual model and modeling deliberations, but assume that a conceptual model may include deliberations in addition to the other kinds of information.

Model execution. Explanation generation technology can be combined with the execution of conceptual models to explain both the external and the internal dynamics of models. This requires, however, that the behavior of the models during execution is well documented in the form of detailed traces of computations and instantiations. Additionally, since executable conceptual models do no include any complete user interface specification, explanations can be used to explain what the system is doing and why certain inputs are necessary.

The two dialogues below illustrate the use of explanations in model execution environments. In the first dialogue, two possible explanations are included, each of them tailored to a specific context and/or user.

user: *"Why was the loan application rejected?"*

system$_2$: *"The requested loan was \$100,000. The loan limit was \$80,000. Using the rule*

> *If* Loan_limit.maximum *is less than* Requested_loan.amount, *then* Recommendation.result = rejected *and* Recommendation.offer *is equal to* Loan_limit.maximum;

it was decided to send Loan_rejection."

system$_3$: *"Because the requested loan was \$80,000, the customer's income \$50,000, her classification* 'neutral', *her total savings \$50,000, and her total loans \$70,000."*

The dialogue below helps the users in the execution of the conceptual model.

user: *"Why do you need* Customer_income*?"*
system: *"The customer was not registered. I need the customer's income to register the customer and to process the loan request."*

In more advanced execution environments, *symbolic* or *hypothetical* executions are available. In symbolic executions, symbolic inputs are used instead of concrete values, and a trace specifying all the possible execution paths is generated. Hypothetical executions are typically initiated by *"What-if"* questions and force the execution environment to investigate the consequences of inputting certain values or using certain alternative model elements in the execution.

6.6 Support for Model Comprehension in PPP

In this section we will use the following example to ilustrate the techniques for model comprehension in PPP. Only some of the process models and one PLD is shown.

6.6.1 Example

A customer of the bank can have several accounts and loans registered. All customers are identified by an id, but the bank also keeps the customer's name, address, and classification. The classification is the bank's judgement of the credit worthiness of the customer, and is assigned one of the values good, neutral, *or* bad. *Customers are divided into two categories, institution customers and person customers. Associated with an institution customer is a set of users, i.e. names of persons authorized to use the customers' accounts. The salary of person customers is also kept. The accounts have an identifying number, a balance, and an interest rate, while a loan is characterized by a loan id, interest rate, date granted, initial loan amount, and balance. Associated with loans is a payment plan that says how they are to be paid*

back in terms of smaller regular payments. A payment is registered with an id, a date of payment, a due date for the payment, and an amount. Transactions on the customers' accounts contain information about the transaction id, the amount to be deposited or withdrawn, the date, and the transaction type (either deposit *or* withdrawal*).*

There are four functions in the systems:

- *The customer can apply for a loan,*
- *open an account,*
- *deposit and withdraw money, and*
- *monthly statements of accounts are sent to all customers on the first day of each month.*

Applying for a loan, the customer must first have opened an account in the bank and cannot have been classified as a bad *customer. The requested loan amount is compared to her salary, current loan, and current bank account balances, and the bank may offer either the requested loan or a somewhat smaller loan.*

If the customer is to open an account for the first time, she must also be registered and given an initial classification as a neutral *customer. If she is already registered, the new account is just added to the existing bank accounts.*

A new transaction is first checked to find its category (deposit or withdrawal), and to see if the specified account exists for the customer. A withdrawal is then checked to see if the balance exceeds the requested amount. If so, the customer gets a notice, and must respond with a lower amount. She can abort the transaction by giving a zero in response to the request. The verified withdrawal transaction is then processed by updating the customers account. A deposit is processed directly. In both cases, information of the processed transaction should be given to the customer, and stored for later use.

PPM Models. A top level PrM model for the bank system is shown in Fig. 6.9.

P1:Process transaction consists of five subprocesses, P1.1 through P1.5, as shown in Fig. 6.10.

A PLD Model. Figure 6.11 shows the process logic of process P1.2 represented in a PLD model.

6.6.2 Overview of Techniques

An overview over the comprehension support techniques in the PPP tool is given in Fig. 6.12. Three different techniques are shown as well their integration. The heart of the architecture is the *PPP conceptual model*. This model can be transformed into an *executable program*, a *model view*, and/or a *explanation*. By executing the program an *execution trace* is generated. The trace contains information about the dynamic properties of the model. Information from this trace is used to produce *explanations*. These explain observed

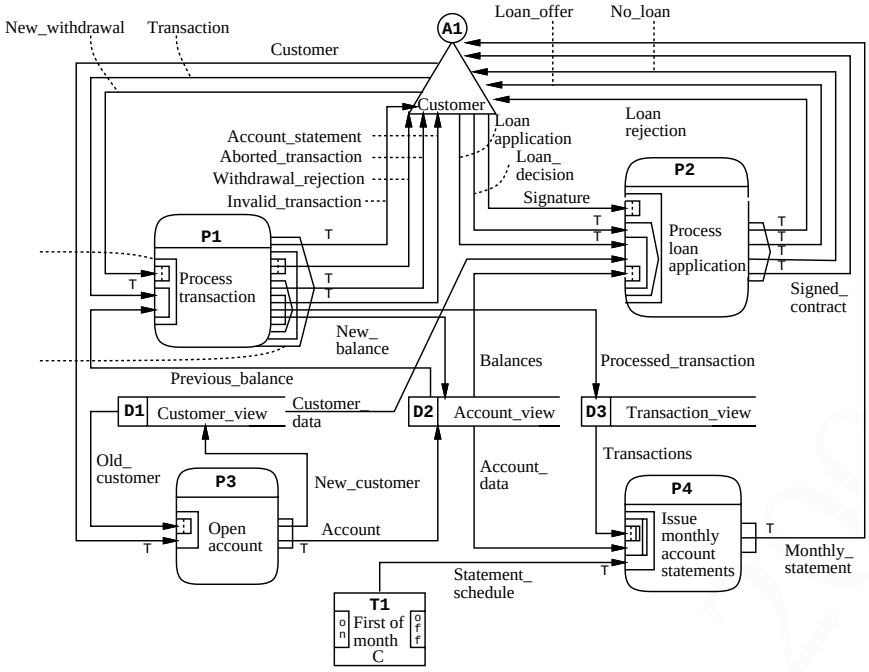


Fig. 6.9. Portions of a PrM model for the banking system

behavior in response to questions posed by users or developers. This explanation module may also exploit views of the model. Combining the pictorial view of the model and the textual explanation, *multi-media* explanations are supported.

We will now present the techniques individually below. The integration aspect will be emphasized by illustrating the various transitions between the supported techniques.

6.6.3 Code Generation

Within the PPP environment several translation assistants has been developed. In addition to the two main assistants with Ada and TEQUEL/C[1] as target languages, the following main translation assistants have been developed in relationship with the PPP environment:

Translating PPP models to Simula/Demos By transforming PPP models to programs in Simula/Demos [158], behavior of real-world phenomena can be simulated. The PPM and PLD languages are slightly extended to accommodate this feature, especially for sampling of values from probability distributions. Using simulation models, non-functional properties

[1] A description of TEQUEL is given in Sect. 6.6.5.

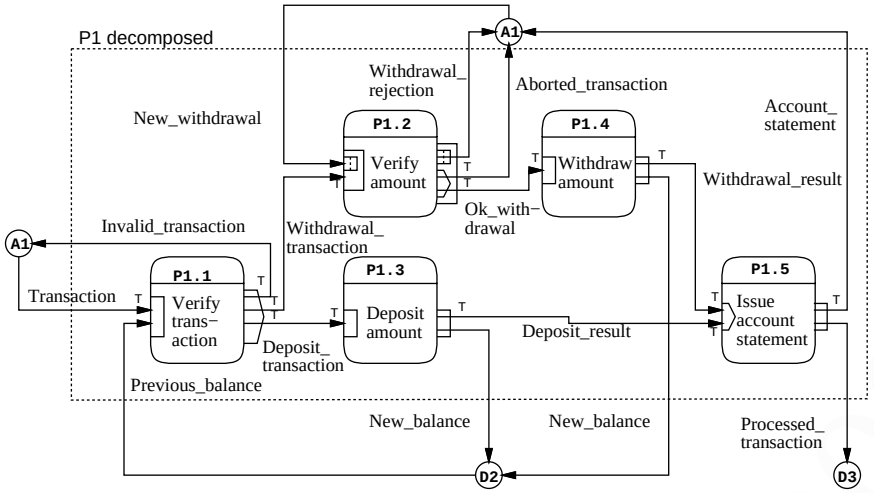


Fig. 6.10. A decomposition of transaction processing

like performance and reliability can be validated. Validation is supported by various traces and statistical reports generated during execution.

Translating ONER Models to SQL A model of the database can be represented in ONER. Going directly from conceptual models to program code, the PPP environment transforms the ONER model to SQL statements that define the corresponding database. No additional statements are necessary to construct the database of the system. Since ONER resembles the traditional ER language, the transformation algorithm is almost similar to the ones used on ER models (see for instance [372]).

Translating PPP/UID Models to C/Motif The user interface model, represented in UID, is transformed to a combination of C code and Motif code [193]. The translated code is complete and runnable, but in the current implementation, it is not integrated with the rest of the environment. Further work in this area is currently taking place. The main motivation for including the transformation is to enable prototyping of user interfaces.

In the sequel, we will concentrate on the two assistants that translate:

1. PPP models to Ada and
2. PPP models to TEQUEL/C

They are presented in Sections 6.6.4 and 6.6.5, respectively. The overall translation strategies are described and some selected translation rules will be included to the extent they shed light on the strategy.

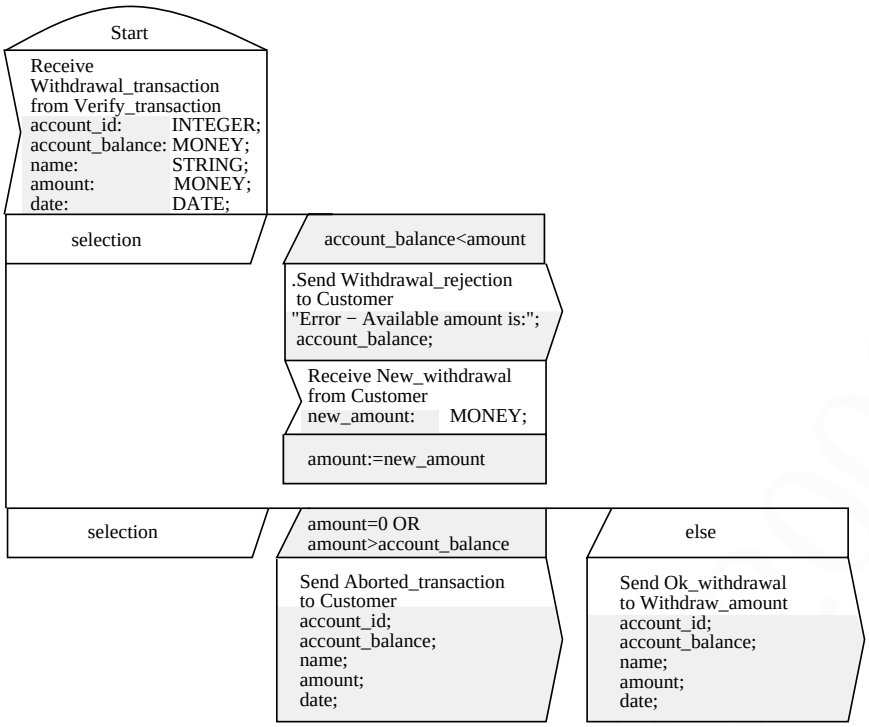


Fig. 6.11. A PLD describing process P1.2

6.6.4 Translating PPP Models to Ada

In this section we will take a closer look at the assistant for translating PPP model to Ada code. The main structure of the translation strategy will be outlined and the validation properties of the generated prototypes are discussed. The following presentation is very much based on [240].

The Overall Translation Strategy. As with all our assistants, the motivation for using Ada as a target language is based on semantic considerations rather than on the need for an efficient implementation of an Ada program. Thus, the semantic correspondence between PPM/PLD and Ada is exploited so that translations easily can be established. The most important arguments for translating PPM/PLD models into Ada can be informally stated as follows:

– PLD diagrams may be executed in parallel, and except for the exchange of data these diagrams run independently of each other. Transforming to Ada, we could let Ada tasks implement the PLD diagrams without any code controlling their execution.

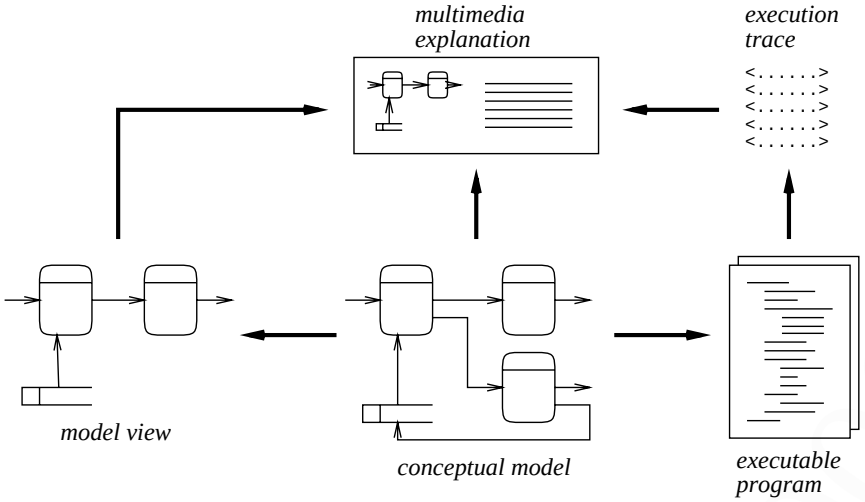


Fig. 6.12. Integrating techniques for the support of model comprehension

1. *For every PLD diagram create an Ada task.*
2. *Sending and receiving data between processes is translated to rendezvous in Ada.*
3. *User communication is achieved by requesting data from the terminal and sending data to the screen.*
4. *Every task contains an outer infinite loop construction and a block corresponding to the PLD model which the task represents. The loop allows the PLD diagram to wait actively for triggering inflows that activates the task.*
5. *Boxes in PLD diagrams are interpreted as actions. Consecutive boxes are implemented as consecutive Ada statements, such that construction* i *corresponds to box* i*. A PLD block appearing to the right of a* choice *or an* iteration *box, forms the scope of the box. It is translated to an Ada block and placed inside the scope of the construction corresponding to the PLD box.*

Fig. 6.13. The overall translation strategy for generating Ada code (From [240]).

– The PLD diagram is a block-structured algorithm that is easily transformed to a structured language like Ada.
– The ports in PPM (and the corresponding constructions in PLD) allow for rather complex patterns of process communication. The rendezvous mechanism in Ada makes it possible to encode the same complex communication patterns for tasks.

These features of Ada made it possible to translate a PPP model to Ada code. The overall translation strategy is shown in Fig. 6.13. Details of the algorithm has been formalized as a set of translation rules. In Fig. 6.14 we have

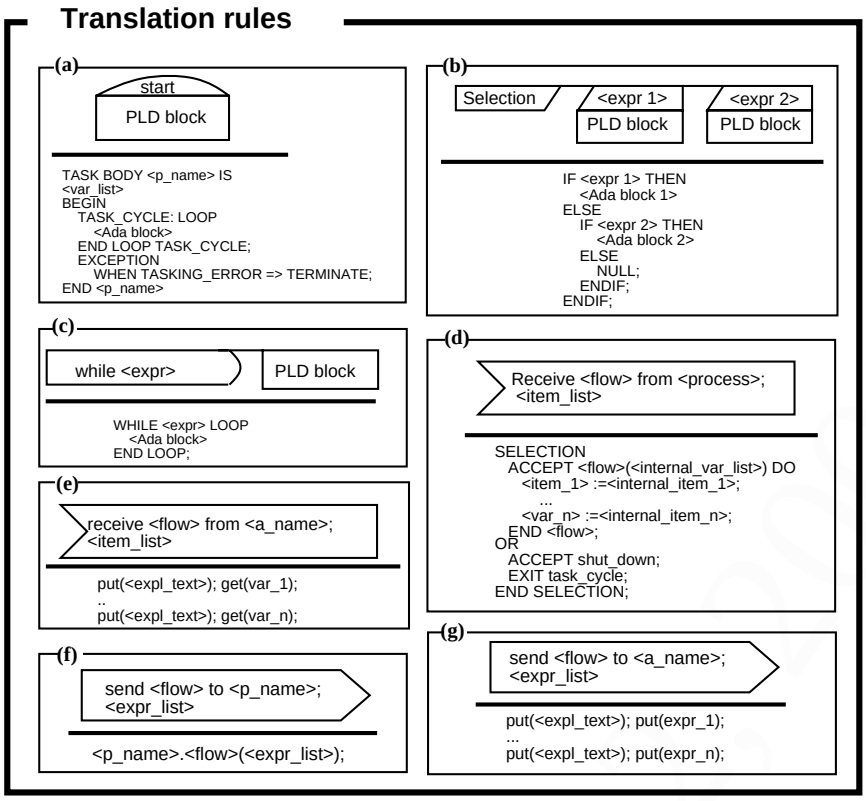


Fig. 6.14. Some selected translation rules the "PPP-Ada assistant" (From [240])

shown some selected translation rules using the graphical notation for such rules from Broy [42]. Different patterns in a PLD model and their associated Ada statements are indicated. The main task of the implemented transformation algorithm is therefore to detect different patterns in the PPM/PLD model that are defined by the translation rules and translate them into their Ada counterpart.

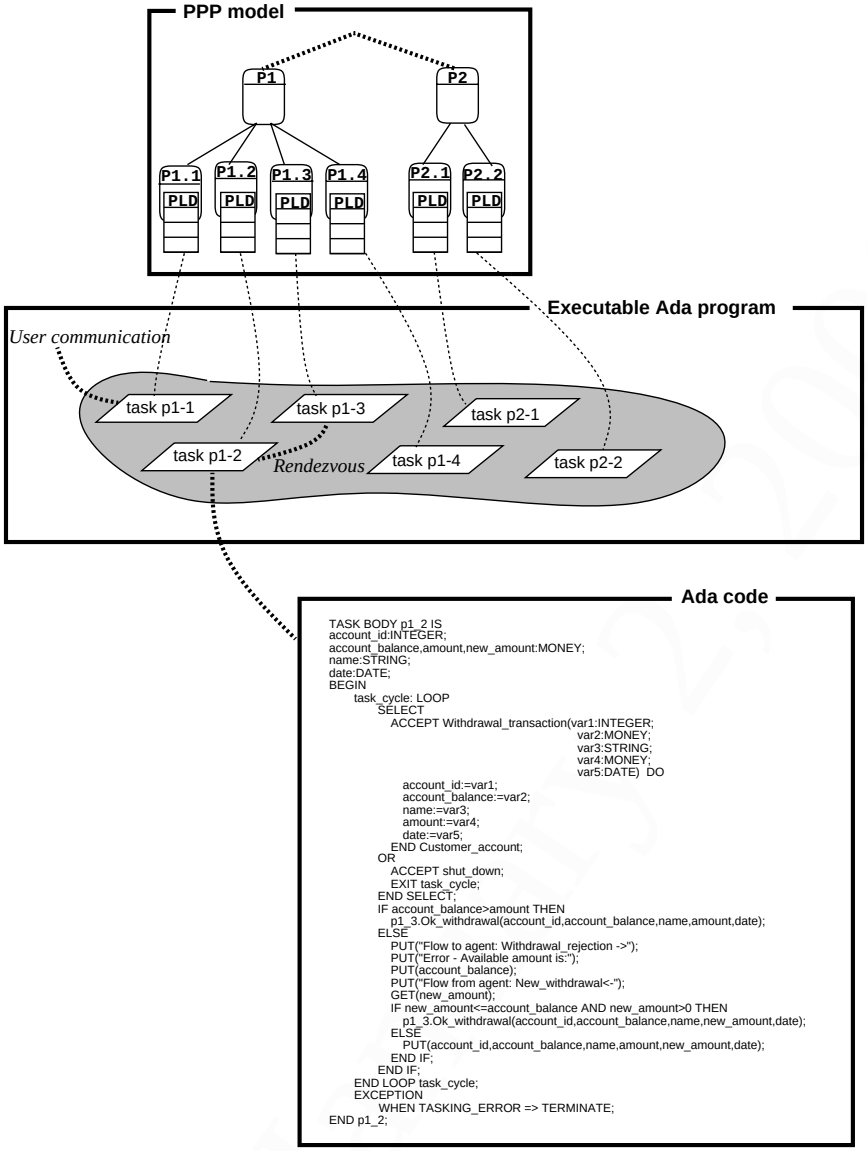


Fig. 6.15. An overview of the Ada translation process

It must be added that the translation schema is a bit more complicated than what the rules in Fig. 6.15 could suggest. In additional to these rules, we have more complicated rules handling complex PLD patterns. In particular, when receive constructs are placed inside the scope of selection or iteration constructs, complex rendezvous statements in Ada can be generated. Anyway, the rules in Fig. 6.15 should give a good indication of how PLD models are implemented as Ada tasks.

Figure 6.15 gives an overall picture of how our PPP model of the bank domain is translated to an Ada program. By following the translation algorithm, we end up with a set of communicating tasks and every task corresponds to a PLD model. The details of the Ada code for the task generated from P1.2's PLD model from Fig. 6.11 is also shown in the figure.

Execution and Validation. Execution of the Ada code relies on the mechanisms provided by the Ada environment. There is no main loop that controls the execution of tasks, so all the tasks are automatically activated as the program is started. When the program is terminated, the terminating task[2] sends a shut-down message that kills all tasks of the program.

In the following session, we will show how the generated prototype may be used to uncover invalidities hidden in the model.

Assume that an early version of process 1.2 and its corresponding PLD model as shown in Fig. 6.16. Here we see that the process has an output port of AND type.

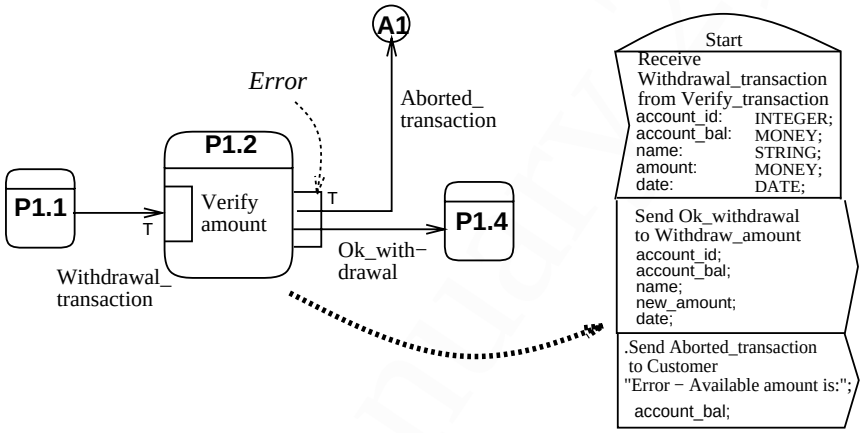


Fig. 6.16. An early version of process P1.2 and its corresponding PLD model

After code has been generated from this model, the dynamic properties of the model can be examined. The first one notices is that the program

[2] Terminating task corresponds to a special process (PLD diagram) defined as 'terminating' in the model.

aborts the transaction regardless of the values of amount and account_bal. Thus, the customer always receives the error-message defined in the PLD model. To correct the error, the output port must be substituted with a new port of XOR type. Doing so, the process will evaluate whether the amount is accepted — ok_withdrawal is sent — or (exclusively) it aborts the transaction — aborted_transaction is sent.

An execution with this error or with the corrected version might reveal that the interaction with the process is limited. That is, the process does not encourage a correction of amount if amount > account_bal. So, a more flexible process that allows for correction of amount during the process execution results in the model shown in Fig. 6.11.

Evaluation. The description above shows the feasibility of translating PPP models to Ada code. The Ada program runs, and its functionality complies with the intended interpretation of the PLD language. In that sense, the generator provides valuable prototypes of the functional properties of the system.

The main weakness concerning the validation properties is that the prototype relies too much on the Ada environment. Different execution mechanisms like step-by-step execution, breakpointing, tracing etc. described in Sect. 6.2 are limited in this environment. Moreover, direct feedback of the execution to the PPP model in form of animation is hampered by incompatibility in the Ada and PPP environments. Thus, the validation of the model will in many ways resemble program testing. One can detect the presence of errors and misconceptions, and hopefully correct them through model inspection. However, one can usually not be certain that all such errors and misconceptions are detected. Still, with the possibility to execute conceptual models, we have better facilities to validate dynamic model properties.

It should be stressed that the generated code is of prototype quality and is *thrown away* after the model has been validated. That is, the translation has ignored design knowledge that affects the efficiency and effectiveness of the code. Particularly, the two most important ones concern the code's *modularization* and *efficiency*. An obvious problem in this respect is the inappropriateness of having all the tasks run concurrently. The generated code is a flat structure of Ada tasks, and it is usually quite difficult to analyze or modify it. This is not a major drawback as long as the program are only generated to prototype the system. But if this prototype is to be modified, e.g. to include a more realistic user interface or to build the real system code, the lack of modularization could hamper the work. Suggestions for meeting some of these problems are given in [150].

The long term goal of supporting the complete life-cycle requires that prototype can *evolve* into the final information systems with acceptable performance. This further requires that more design knowledge is represented as translation rules, so that a modularized and efficient system can be gen-

erated automatically, or semi-automatically with guidance from the system developer.

6.6.5 Translating PPP Models to C and Prolog

In this section we will take a closer look at the assistant for generating C and Prolog code from a PPP model. This approach applies a temporal rule manager built on top of standard prolog, and a parser for a programming language for representing temporal rules to be applied by the rule-manager. This programming language has been developed at Imperial College, London [293] in the scope of Tempora. The rules are on the form [212]:

formula about the future <= formula about the past

These rules are evaluated with respect to a particular state in a *temporal database*, yielding a number of formulae about the future which must be made true, if not already true.

The main structure of the translation strategy will be outlined and the validation properties of the generated prototypes are discussed. The interested readers is referred to [206, 237, 238] for a more complete documentation of the translation assistant.

The Translation Strategy. In general, Prolog is used in this approach for the representation of the database, timers, and the temporal relationships between the processes. The processes represented as PLD's on the other hand, is represented as C-functions.

User View of the Execution and Validation. User involvement during the execution is central for model validation. In connection with the work presented here, we have identified three major tasks where the user can participate: (1) setting up the execution session, (2) viewing the execution trace, and (3) inspecting the temporal database. The tasks are briefly explained in the sequel.

Setting up the Execution Session. An execution session is set up by defining the test data. That is, the initial database content and the external events which are invoked during execution. Both aspects should reflect the situation in the problem domain.

We here assume that the initial database content is as indicated in Fig. 6.17a. For simplicity reasons, we have only shown the information that is relevant for processing a transaction. Thus, the database is loaded with one customer, Odd Ivar, who has account account 1001 with balance 5612.

Here, we define the execution to last for 10 ticks (here minutes) which should cover the period of processing a transaction. During the period the user invokes the external events that are shown in Fig. 6.17b. At minute 1, Odd Ivar issues a withdraw transaction on account 1001 with amount 6000. As will be evaluated, the account number is wrong and a new transaction

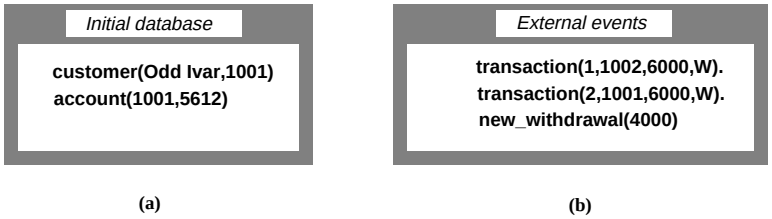


Fig. 6.17. Test data for the execution session

(2) is issued at minute 3. This time the account number, 1001 is correct, but the amount is too high compared to the available amount. Odd Ivar is given a last chance and issues a new_withdrawal transaction amounted to 4000 at minute 6. This will be accepted.

The external events describe an execution session. In the following we will explain how the user can interact with the Rule Manager during the execution.

Viewing the Execution Trace. The Rule Manager executes the generated prototype on the basis of the initial database content and the external events. Here, the Rule Manager uses a textual interface for user communication. The execution trace shows the situation at each tick and allows for invoking external events.

In Fig. 6.18, the execution trace of our example as it is produced by the Rule Manager is shown. External events are entered at the ticks (minutes) described above. Moreover, the internal events like triggering and terminating flows are indicated as well as the actions — execution of the process. Response to agents is given as plain text.

Inspecting the Temporal Database. In addition to the execution trace, the Rule Manager provides a temporal database throughout the execution. Roughly speaking, the content of this database represents the historic view of the execution. By inspecting the temporal database, it can be learnt within what time period the collected execution information has been valid.

In Fig. 6.19, we have shown some of the tuples of the temporal database after the execution. Each tuple is *time-stamped* with a start-time and an end-time indicated in which time period of the execution the tuple was valid. The initial database from Fig. 6.17a is recorded in the tuples 1 and 2. These tuples are permanent during the execution. This is indicated by start-time and end-time being 0 and 99999, respectively.

Furthermore, the temporal database records external events, internal events, and actions where the start-time and end-time collapses into a time-period of one tick. An external event is recorded with the happened tuple 3, 6 and 11, whereas an internal event is indicated in tuple 10. An action is shown in tuple 4.

The tuples 2 and 16 illustrate how the account is updated after the transaction has passed the different checks.

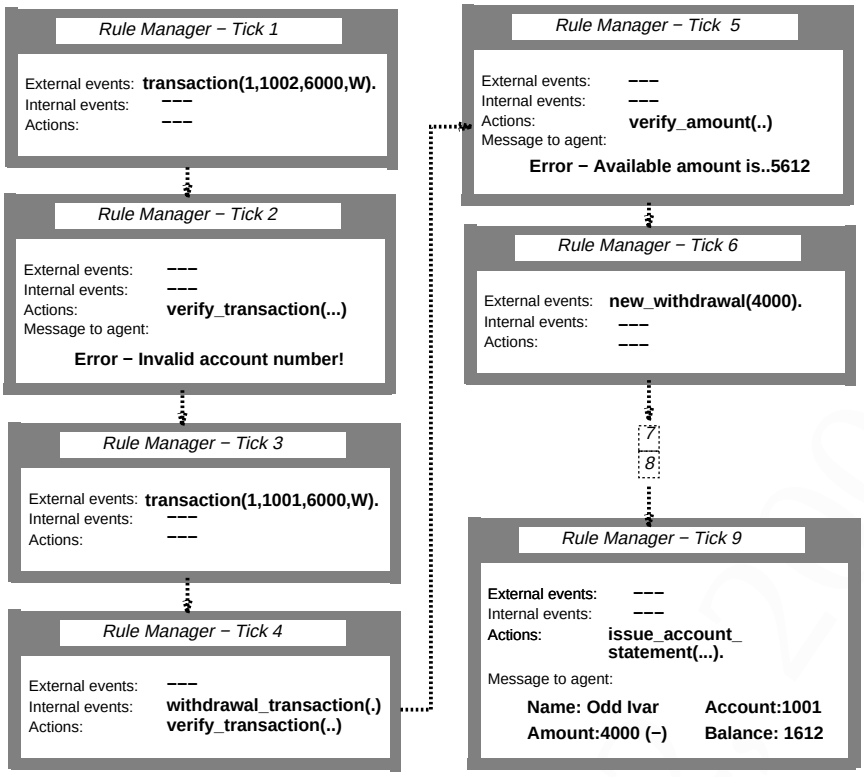


Fig. 6.18. Execution trace of the example

Discussion. The chosen approach enables the user to validate model behavior based on executing the generated prototype. How well validation can be supported depends on several factors. Here, we will discuss the validation potential with respect to (1) the modeling language and (2) the tool support. The first factor depends mainly on the PPP environment, whereas the last is dependent both th PPP environment and the Rule Manager. In the sequel we will briefly discuss these factors with respect to the current status and devise possible improvements.

The Modeling Languages. In PPP as presented here, the internal logic of automated processes is specified by PLD models. Another possibility is to use DRL-rules for the process description. A translator from Tempora ERL to Prolog has been developed, and how to include the deontic extensions of DRL have been outlined in [207].

The UID language can be used to specify the user interface for window applications. By exploiting UID within the frame of this work, the external events and execution results can be related to a generated window. As such, the execution can be presented in a way that is more like the one the user will

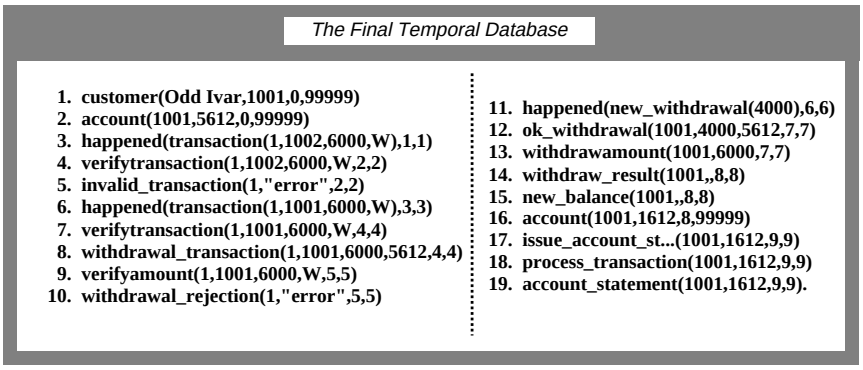


Fig. 6.19. Parts of the temporal database after execution

meet in the final application. By combining this facility with the prototyping approach, *multi-modal validation* is enabled, showing the same situation through several viewpoints.

Tool support. In the current version of the tools, the PPP tool and the Rule Manager are loosely coupled. The validation is therefore carried out by remotely comparing the model behavior with the execution trace. By *data integration* [369], the tools will interpret the underlying data structure equally. Such an integration could have interesting effects on the validation support. First, one could envisage that the Rule Manager would be an integral part of the PPP tool and is directly invoked from PPP, whenever an execution is appropriate achieving *control integration*. Secondly, faster generation of the executable code can be achieved by retranslating only those parts of the model that are modified. Finally, *animation* of the model can be provided by directly feeding the execution trace and the temporal database into the model. For instance, events and actions could be shown by highlighting the actual PPM construct. Moreover, external events can be entered in a "pop-up" window and temporal results of the execution could be accessed by "clicking" on the relevant PPM constructs.

By presenting the dynamics of the simulation graphically, the user may get a deeper understanding of the behavior of the model. We envisage that all the events in the model are presented sequentially. Moreover, active elements are illuminated and when an item is put on a flow it will be shown on the screen. Thus, every state change in the model will be shown.

Independently of data integration, the presentation of execution traces and the temporal database can directly be improved by new versions of the Rule Manager. From providing a textual user interface as described in this section, the new versions have adopted a graphical interface under X windows. Thus, a more flexible and user-friendly execution mechanisms could be supported by upgrading the PPP tool correspondingly, obtaining *presentation integration* [369] of the tools.

The current version of the Rule Manager exploits *step-by-step execution* and *batch execution*. When using step-by-step execution, as shown in Fig. 6.18, *break points* are included at each tick in order to suspend the execution. Then, the user can interactively participate in the execution by invoking external events. Moreover, he can investigate the state of the temporal database. Batch execution is realised by loading the temporal database with initial permanent data plus the events that shall happen during the execution. It is also possible to combine these two execution types. Some parts of the execution can be run in a step-by-step fashion and other parts in batch.

Also, the code generator provides a flexible way of specifying the "execution speed" or the tick-length. For instance, to simulate various real-time situations, a time granularity of a tenth of a second may be appropriate, whereas a granularity of minutes was suitable for transaction processing part of our bank example. Finally, the execution can be triggered to run for as many ticks as wanted.

6.6.6 Filtering in PPP

In PPP, simplifications are provided by allowing different views of a model, each focusing on a different aspect of the system. A view is defined by a set of explicitly defined abstraction mechanisms, manually produced by a developer, or a combination of the two. Rather than operating on a full model, relevant views can be applied at different stages of the development process depending on the problem to be solved and the actors involved. In this way, a systematical approach for suppressing irrelevant details and highlight relevant details of a model is provided.

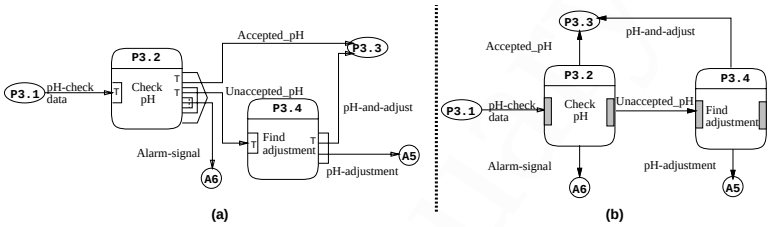


Fig. 6.20. (a) A model view resulting from a component abstraction (b) Ports abstracted away and layout is improved

To illustrate the features of abstractions, two simple abstractions are described below. More details about the abstraction-based approach are given in [331].

Given a situation where we want to check different pH values and then concentrate on those values which demand that adjustments are found. This means that mainly two processes of Fig. 6.20a are of interest, namely `P3.2`

and P3.4. A model view generated based on the a *component abstraction*. Furthermore, the details of the ports are superfluous and only clutter the diagram. In Fig. 6.20b, the ports are abstracted away and the diagram is restructured to exploit the relaxed spatial requirements.

Views produced by the abstraction facility can be utilized by the explanation facility. The routine

```
display([<element>],[<views>])
```

is made available to the explanation generator which specifies a total view containing [<elements>] that is a combination of the predefined subviews [<views>]. When the abstraction facility is invoked by this routine, a view of the conceptual model will be displayed to the user. In this way, an explanation of the relevant part of a specification can be visually shown together with the textual explanation.

6.6.7 Execution, Tracing, and Explanation Generation in PPP

The overall architecture of the explanation component and its interconnections with other components of the PPP environment has been depicted in Fig. 6.21, and will be introduced in the following.

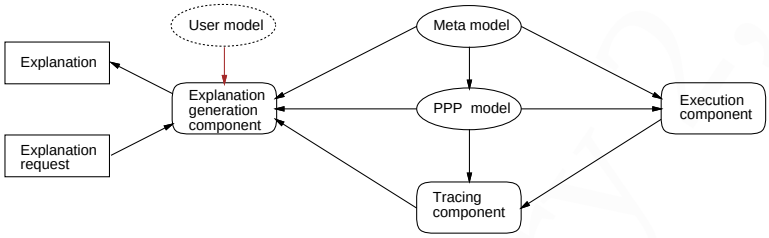


Fig. 6.21. Architecture of the explanation generation system

Using the PPP metamodel and conceptual models written in the PPP language, the *execution component* transforms these models to an executable representation. The resulting code is then executed, exposing the users to the dynamics of the conceptual model. During execution, the users simulate the system environment, giving inputs and receiving outputs as requested and produced by the executing model.

In such a session, users have the opportunity to interrupt the execution, and issue a request for explanation of the observed behavior. These requests are then interpreted by an *explanation generation component*, which responds by generating an appropriate explanation. The generation is performed in two subphases, first one which determines the content and structure of the explanation, and then a phase which produces suitable presentations for user

communication. The content of the explanation is found by selecting information from different sources. First, the metamodel and the conceptual model are both used. Second, the selected content may be influenced by the characteristics of the users interacting with PPP, and by the context for the explanation request. It is then possible to tailor the explanation to the receiver, for instance to provide different explanations to end-users and system developers. Additionally, the explanation must refer to the events which actually happened during execution of the conceptual model. This information is provided by a *tracing component*. It records occurring events as reported by the execution component, according to a predefined trace schema. The interface to the explanation generation component consists of a set of trace query functions, which provide views into the recorded trace, tailored to commonly occurring explanation requests.

Drawing upon these different sources of information, the explanation generator constructs an explanation represented in an explanation modeling language. The resulting explanation may subsequently be translated to a multimedia presentation, suitable for user communication.

Model Tracing. The tracing component as shown in Fig. 6.21 is divided into two subcomponents; a *reported events handler* which interfaces the execution component and stores information about reported events according to a predefined schema, and a *query handler* which offers the explanation generation component high level queries tailored to commonly occurring explanation requests.

Reporting Occurring Events from the Executing Model. In order to provide comprehensive explanations of observed behavior from an executing conceptual model, tracing techniques have to be employed. The question is what kind of information has to be recorded. To answer this question, and hence provide a schema for trace representation, we have taken a general view of the statics and dynamics of information systems.

We consider a model to pass through a sequence of states during execution. A state is the aggregated value of all state components in the model. In PPP these components are flows, stores, and local variables of PLD's. Additionally, states are associated with a state number, or a time point, giving their temporal ordering. Various types of events may bring the model from state to state during execution. For instance, a model can change state through interaction with its environment. Such events correspond to input flows from external agents. After such an event, the executing model responds with applying a series of transformations, or *dynamic rules*. In the course of these applications, the model passes through a series of states until a new equilibrium is reached. It is information about environment interaction and application of dynamic rules that have to be recorded in a trace.

Conceptually, a trace can be considered a directed graph, where the nodes correspond to states, and the edges correspond to events of different kinds, i.e. interaction with the environment, and application of dynamic rules. We

have defined a trace schema based on the various events which may occur. Since this schema is derived from the edges in the trace graph, information about the nodes (i.e. the states) is only implicitly represented. Common to all events is that their temporal ordering is kept by storing a reference to the 'head' state and the 'tail' state of their edge in the trace graph. The events must be uniquely identifiable by name or other means of direct identification. Additionally, changes made to the state are recorded for all events which affect the state directly. A change is represented by a reference to a state component and its new value.

Now, we can present the most important parts of the schema informally as follows:

External events correspond to triggering input flows from external agents to processes. Such events may occur spontaneously, and are not under the control of the executing model. Information stored about such events are a unique reference and effects on state components.

Other inputs, i.e. non-triggering input flows from external agents are recorded similarly, except that they may have an associated precondition. A preconditon refers to state components and their values, so a list of the components and values referenced has to be included.

Complex dynamic rules (transformations) are recorded with reference and precondition values. In PPP, these correspond to processes on all levels of decomposition, and to the PLD constructs 'selection', 'alternative', and 'loop', i.e. those which initiate construct blocks in a PLD. A typical precondition in PPM would be "a triggering flow is received", and a block in PLD initiated with an alternative construct would have a precondition corresponding to the expression in the construct. Note that complex rules change state only indirectly, through application of sub-rules, so their effects on the state are given implicitly through these sub-rules.

Simple dynamic rules update one or more state components directly in an atomic operation. Assignments, sendings and receivals of data in PLD are considered simple dynamic rules. Information about the changes they cause on state components are included in the trace. Also, values of state components referenced in preconditions, or referenced as new values of state components are computed, are included as well.

Having defined a trace representation, we have to provide mechanisms for storing information from the executing models. In order to do this, the model has to be instrumented with *probes* at the appropriate locations. These probes are calls to procedures which record information about occurring events in the trace. The reported events handler offers reporting procedures to be called from executing models. The reported events must comply with the schema presented above, but the amount and type of information to be recorded depend on the needs for different kinds of explanations. The procedures correspond to the various events identified above, hence we have for instance a

reporting procedure *reportexternal* which takes two arguments, a reference to an external event, and the direct effect on state components. The state numbers needed are inserted before storing the information in the trace. For events ranging over multiple states, it is possible to report their initiation and termination separately.

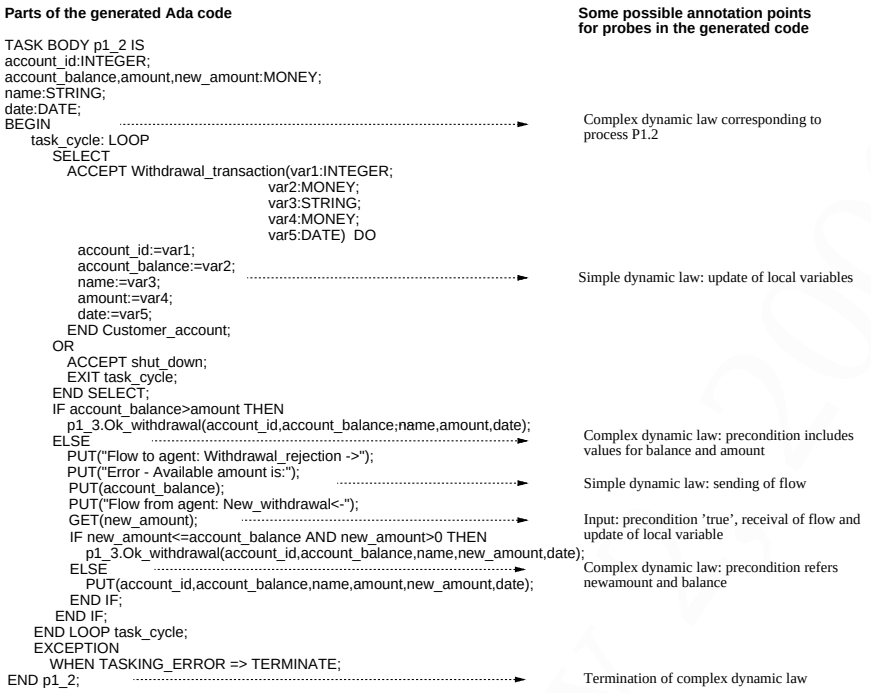
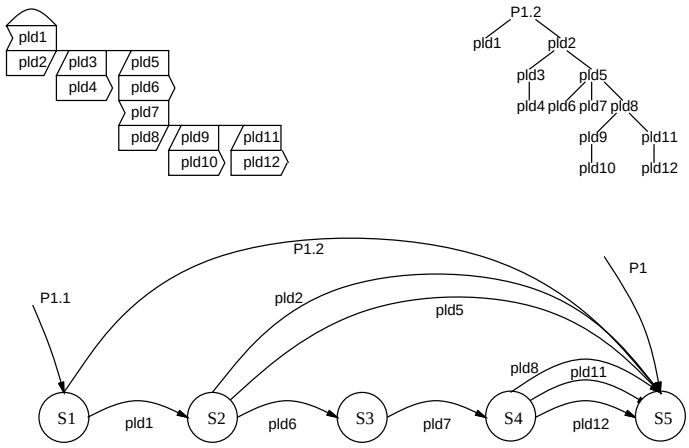


Fig. 6.22. Generated Ada code from a PLD indicating the insertion of probes

Figure 6.22 indicates how probes may be inserted by the Ada code generator. Figure 6.23 shows a small portion of a trace graph corresponding to an execution of process `P1.2: Verify amount`, where the requested amount to be withdrawn exceeds the balance of the account. The customer is then asked for a new amount, but responds by aborting the transaction. The nodes represent states with a state number, and the edges represent the events from the execution. The figure also shows the representation of some selected portions of the trace, corresponding to the probes in Fig. 6.22. The order of the elements in the tuples is: Reference, values of state components in precondition, from-state, to-state, and state change. The hierarchical structure of dynamic rules derived from the PLD is shown as well, to illustrate the relationships to the trace graph.




```
internalsimple(pld1,[(Withdrawal_transaction,(jon,100,121292,80,1234))],S1,S2,
[(name,jon),(amount,100),(date,121292),(account_balance,80),(account_id,1234)])
internalcomplex(pld5,[(account_balance,80),(amount,100)],S2,S5)
internalsimple(pld6,[(account_balance,80)],S2,S3,[(Withdrawal_rejection,("Error - Available amount is:",80)])
inputevent(pld7,[(New_withdrawal,0)],S3,S4,[(new_amount,0)])
internalcomplex(pld11,[(new_amount,0),(account_balance,80)],S4,S5)
```

Fig. 6.23. A small portion of a trace graph for execution of the PLD of `P1.2`

Retrieving Trace Information for Explanation Support. We have defined a set of *trace views* which provide necessary retrieval mechanisms for explanation generation support. The views are tailored to explanation strategies for commonly occurring explanation requests. Each view is associated with a query which can be made to a query handler. The queries can be nested, providing combinations of views. The views defined so far are as follows:

In the **State component view**, the focus is on changes made to a specified state component. Information about the state change is retrieved as it is stored in the trace. The query specifies the component and other conditions to be satisfied for the change to be retrieved. For instance, to find the last sending of the flow `Withdrawal_rejection` the query *FIND LAST CHANGE FOR Withdrawal_rejection*[3] can be issued. The retrieved information could be the third tuple extracted from the trace shown in Fig. 6.23. Further conditions can also be specified, for instance conditions on the values of the changed state components.

The **External event view** is used to trace the interaction with the environment. A query can be made to retrieve information about specified external events and the changes made to state components as a result of these.

[3] The syntax presented should not be taken too literally. The interface to the explanation generator is a set of function calls corresponding to each query.

In the **Rule application view** focus is on a particular dynamic rule, and information about applications of this rule can be retrieved. For instance, *FIND ALL APPLICATIONS OF P1.2* would retrieve information about executions of this process for a specified period of execution.

The **Rule context view** provides the context of a rule, which includes the set of super-rules executing at the same time. Referring to Fig. 6.23, the context of `pld6` is `pld5`, `pld2`, and `P1.2`. A nested query must be made to specify a particular application. So, the query *FIND SUPER OF (FIND LAST APPLICATION OF pld6)* can be used to retrieve information about the PLD block initiated with the alternative construct `pld5` ("Amount is less or equal to account_balance"), and hence support explanation of why the flow `Withdrawal_rejection` was sent. An alternative query to retrieve the same information would be *FIND SUPER OF (FIND LAST CHANGE FOR Withdrawal_rejection)*.

The **Refers-to view** tracks the dependencies between dynamic rules. For instance, if a precondition refers to state components changed by other dynamic rules, the rule refers to those rules. In Fig. 6.23, we can see that the sending of `Withdrawal_rejection` depends on the value of `account_balance`, changed by the receival of `Withdrawal_transaction`. Thus, the query *FIND REFERS (FIND LAST CHANGE FOR Withdrawal_rejection)* will retrieve information about the receival of the flow (`pld1`), i.e. the first tuple extracted from the trace in Fig. 6.23.

From the **State view** it is possible to derive information about states from the trace, for instance which rules terminated, which were applied, values of state components etc.

Combinations of views are also possible. We have already indicated how nested queries can be made to combine views. Using the query from the refers-to view above, we can form another query to retrieve information about all changes made to the referred state component (`Withdrawal_rejection`) in a specified period of execution: *FIND ALL APPLICATIONS OF (FIND REFERS (FIND LAST CHANGE FOR Withdrawal_rejection))*.

In addition, we provide the possibility to specify the interval of interest for the validation session. This can be done by calling to procedures which set the start and end of the interesting interval, respectively. The end points are specified by characterizing the states uniquely. For instance, the specification *FROM (FIND LAST STATE WHERE EXTERNAL EVENT)* sets the start of the interval to the last state where an external event took place.
The interface to the explanation component is made up of a set of query functions corresponding to each of the views above. Together they provide retrieval mechanisms which can be used to support explanation generation.

Explaining the Execution of Conceptual Models. For our purposes, we have seen it advisable to divide the process of explanation generation into two consecutive subprocesses, *deep generation* and *surface generation*.

In the PPP CASE environment, a deep explanation generation component has been implemented. It is meant to support the whole process of conceptual modeling, starting from the learning of a modeling language to the validation of the resulting conceptual model. More specifically, the following four areas are to be supported:

Language help facilities Using a simple meta model of the PPP language, the component can explain how constructs are used syntactically and also indicate some of their semantic aspects. Taking a process in PPM as an example, we can generate explanations to questions like *"What is a process?"* and *"How are processes used?"*

Syntax checking Although not a part of the syntax checking module, the explanation generation component provides an interface to the module's messages. Running a verification check on an illegal PPM model, for example, the system could generate the message *"The diagram is illegal, since* `P3` *does not produce any flow. A process must produce at least one flow."*

Model inspection Properties of the conceptual model have been paraphrased and explained in several CASE-related systems (see [77, 316, 358]). Due to the formality of many modeling languages and the size of the corresponding conceptual models, explanations have been deemed useful for people not familiar with reading conceptual models. The model inspection technique serves as a validation technique, and in our component we can tailor the explanations to different user groups, depending on their knowledge and preferences. *"What is process* `P1.2 Verify_amount` *doing?"*, for example, could spark the generation of explanations at several levels of abstraction, using different foci and different terminology.

Model execution When executing PPP models, we can help the users understand the execution by providing *history explanations* and *input justifications*. A history explanation exposes the internal behavior of the model and is generated to let the user validate the reasoning leading to the current results. As will soon be shown, the question *"Why was my withdrawal rejected?"* (referring to flow `Withdrawal_rejection` from process `P1.2`) is a request for a history explanation.

An input justification explains why the model needs certain inputs from the user. They are generated as responses to questions like *"Why do you need* `New_withdrawal`*?"* or *"What is* `New_withdrawal used for`*?"*

In the following, we will discuss the principles of our deep generator with respect to execution-related explanations. In our approach, then, we draw on the results from expert systems, but we have been forced to modify their explanation strategies to cope with the increased complexities of conceptual models.

Deep Generation in the PPP Environment. The deep generation component includes various sources of knowledge and a number of strategies for selecting and structuring elements from the sources to form deep explanations.

The content of the deep explanation is taken from the meta model defining the PPP language, the PPP conceptual model, and the trace from executing the PPP model. These three representations together are referred to as the *source model*. An additional user model and a context specification are used to govern the generation process, but in the presentation to follow we will ignore this tailoring of deep explanations to user and context. All the relevant informations are represented using an explanation modeling language called *EML*, which is based on the attribute-value structures found in many natural language grammars [76]. An *EML* structure, then, contains two parts $\alpha : \beta$, where α is a *characterization* predicate and β an attribute-value structure. The characterization is a classification of the information contained in the structure, whereas the attribute-value structure specifies everything from the user's knowledge of it to its linguistic realization. In this presentation, we will ignore the use of these attribute-value structures and rather focus on the way characterizations make it possible to generate deep explanations. In Table 6.1, we have shown the *EML* characterizations of some of the elements from our PPP model. In the characterization `s(purpose(p1.2))`, for example, `s` says that this is structural information (as opposed to `c` for constraint informations), and `purpose` is a generic relationship with argument `p1.2`.

Table 6.1. *EML* characterizations for some elements of the PPP conceptual model

EML *characterization*	*PPP model informations*
`s(data_flow(withdrawal_rejection))`	declaration of flow `withdrawal_rejection`
`s(generate(pld6,withdrawal_rejection))`	`pld6` generates `withdrawal_rejection`
`s(purpose(p1.2))`	`P1.2`'s purpose is to verify amount

For each source model phenomenon to explain, a specific explanation strategy has been formulated. This strategy determines which elements to include in the explanation, and how these elements should be structured. Interestingly, it has turned out to be possible to define strategies in terms of substrategies and relate their definitions to general theories of text linguistics. In our system we have implemented the strategies for explanation generation as *plan operators*, which are specializations of the relations in Mann and Thompson's general Rhetorical Structure Theory [252]. By using one of these operators, an explanation request, or a discourse goal in general, is decomposed into a number of subgoals or references to the source model. The structures of some of these plan operators, which are also represented in *EML*, are given in Table 6.2. In the first of them, `cause(Value)` is a characterization predicate that refers to that particular plan operator. `Cat` is a category, whereas `head` is the discourse goal fulfilled by using the operator. `Nucleus` and `satellite` are subgoals of `head` in the sense that `head` makes explicit the relationship between them and is fulfilled by fulfilling the subgoals. The subgoals themselves are either characterizations of other operators or characterizations of source model elements. A subgoal may be a

simple characterization (like the nucleus of *cause*), a choice between characterizations (like the satellite of *instantiation*), or a list of characterizations (like the nucleus of *behavior*). If a list of characterizations is specified, the system tries to include in the explanation as many as possible of the list elements. At last, `precondition` includes a number of predicates, either referring to source model elements or forming rules analyzing the source model, and these bind the variables of the operator and constrain the use of it.

To generate a deep explanation, an appropriate discourse goal is picked and the corresponding plan operator decomposed into its subgoals. Using a standard unification-like planning algorithm, the generator expands the subgoals until all unexpanded subgoals refer to elements of the source model. The precondition part of the operators, and also the user model and the context specification, determine which operators to use in this planning process. In the resulting deep explanation, the leaf nodes are content elements that can be transformed to predications in Dik's Functional Grammar. In recent years there have been several promising attempts to produce natural language sentences from these predications, and the surface generator to be added will draw on these experiences.

Generating a History Explanation. Consider the trace in Fig. 6.23. The customer tries to make a withdrawal of 100, but the balance of her account is only 80. Receiving an error message from the system, she might request a history explanation as shown below.

system: Error — Available amount is 80.
user: Why was my withdrawal rejected?

Ignoring the use of features to specify linguistic informations, user properties and explanation contexts, we can explain the generation of deep explanations as a general top-down planning process. Before generating the deep explanation, however, we need to establish the interface between goals in plan operators and informations in recorded trace tuples. Basically, this is done by defining functions in the explanation generation component that correspond to subsets of returned tuples from trace queries. When one of these functions are called, its parameters are translated to forms suitable for trace queries, a specific query is performed, and parts of the query result are translated back to the *EML* formalism. In Fig. 6.24, we have indicated how three of the component's functions are related to queries from the tracing component. The function `get_last` returns an *EML* identifier for the last generated instance of a model element. Function `get_beh` returns the model statement instance responsible for the generation of a specified instance value, and `pre_inst` returns the instances last used as preconditions for the given model element.

To generate a text that explains the sending of flow `Withdrawal_rejection` we choose to invoke the *cause* operator. This operator is used to justify a given value by describing the way model elements are used to generate or calculate that particular value. Prior to the invocation of the operator, though, the operator's input parameter `Inst` is computed from the formula

Table 6.2. Plan operators needed to generate the history deep explanation.

cause(Inst) :	TYPE	operator
	HEAD	cause
	NUCLEUS	s(generate(C,Inst))
	SATELLITE	instantiation(Model)
	PRECONDITION	find_arg(agens,Inst,Inst_ag) type(Inst_ag,Ag) find_last_pre(Ag,Model)
instantiation(Model) :	TYPE	operator
	HEAD	instantiation
	NUCLEUS	behavior(Model)
	SATELLITE	s(pre_inst(Inst_set)) cause(pre_inst(Inst_set))
behavior(Model) :	TYPE	operator
	HEAD	behavior
	NUCLEUS	s(trigger([_,Model])) s(is_precondition_for([_,Model])) s(receive([Model,_])) s(terminate([_,Model]) s(generate([Model,_]))

Table 6.3. Additional operators for generating the input justification.

current_activity(Inst) :	TYPE	operator
	HEAD	current_acitivity
	NUCLEUS	future_use(Inst)
	SATELLITE	super_activity(Activity,_)
	PRECONDITION	s(Element([Inst])) s(decomposition([Activity,Element]))
super_activity(Activity,Super) :	TYPE	operator
	HEAD	super_activity
	NUCLEUS	s(purpose(Activity,_))
	SATELLITE	s(decomposition([Super,Activity])) s(purpose([Activity,_]))
future_use(Inst) :	TYPE	operator
	HEAD	future_use
	NUCLEUS	cause(Inst)
	SATELLITE	behavior(Next)
	PRECONDITION	s(Element([Inst])) s(trigger([Element^termination,Next]))

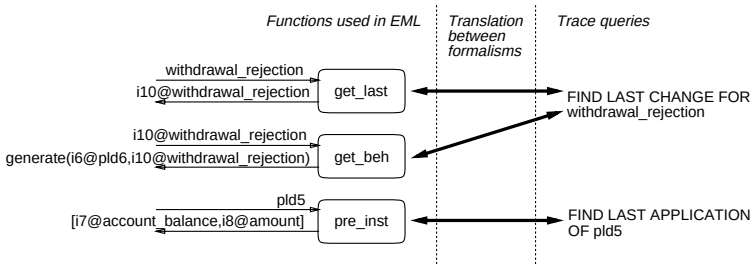


Fig. 6.24. Functions defined to request informations from the trace

```
Inst = get_beh(get_last(withdrawal_rejection)) =
generate(i6@pld6,i10@withdrawal_rejection).
```

In the expression above, `i6@pld6` is the identifier of the last execution of statement `pld6` (see Fig. 6.23) and `i10@withdrawal_rejection` the identifier of the generated flow. *Cause*'s precondition is now evaluated to find variable bindings that make all its predicates true. In this particular operator, all predicates are rules working on the source model, and the first of them returns the agens parameter of relation `Inst` (semantic roles like agens are defined in the conceptual model). The next one determines the type of this agens instance, and the third and last one checks the model to find the last model element used prior to the flow generation statement that has a precondition associated with it. All the predicates in the precondition were found to be satisfiable here. After a successful binding of variables used in `precondition`, the operator's `nucleus` and `satellite` are instantiated, and we get the instantiated *cause* operator structure in Table 6.4.

Table 6.4. Instantiated *cause* operator.

TYPE	operator
HEAD	cause
NUCLEUS	s(generate(i6@pld6,i10@withdrawal_rejection))
SATELLITE	instantiation(pld5)
PRECONDITION	find_arg(agens,generate(i6@pld6,i10@withdrawal_rejection),i6@pld6) type(i6@pld6,pld6) find_last_pre(pld6,pld5)

In the instantiated operator, the precondition has found that `i6@pld6` is the agens of `Inst` and `pld6` the type of `i6@pld6`, and that `pld5` has the precondition governing the execution of PLD construct `pld6`. The instantiated nucleus is a source model reference that is to be realized as a natural language clause. The satellite, however, refers to the *instantiation* plan operator, and

the generation process proceeds by instantiating and including this operator into the structure.

The *instantiation* operator does not have any precondition. Its nucleus is instantiated with the value `behavior(pld5)`, and since there are two alternative satellite values, the system just picks the first one and instantiates it. If it later turns out that it is impossible to instantiate the satellite, or perhaps impossible to decompose or realize it, the system backtracks and tries the other alternative. Instantiating the satellite, the generator calls the function `pre_inst` to request a set of instances from the trace. The resulting satellite is `s([i7@account_balance,i8@amount])`, which is a simplification of the two source model references `s(structure([i7@account_balance]))` and
`s(structure([i8@amount]))`.

Instantiation's nucleus is expanded using the *behavior* operator, which contains neither a precondition nor a satellite. The nucleus is a list of source model references, and the system tries to instantiate and include as many as possible of these informations. Since `pld5` is an alternative construct, only the reference

`is_precondtion_for('account_balance < amount',pld5)`

is found in the PPP model, and we get the final deep explanation structure as shown in Fig. 6.25. All the leaf nodes of the structure are content elements that are supposed to be mapped into natural language in the surface generator.

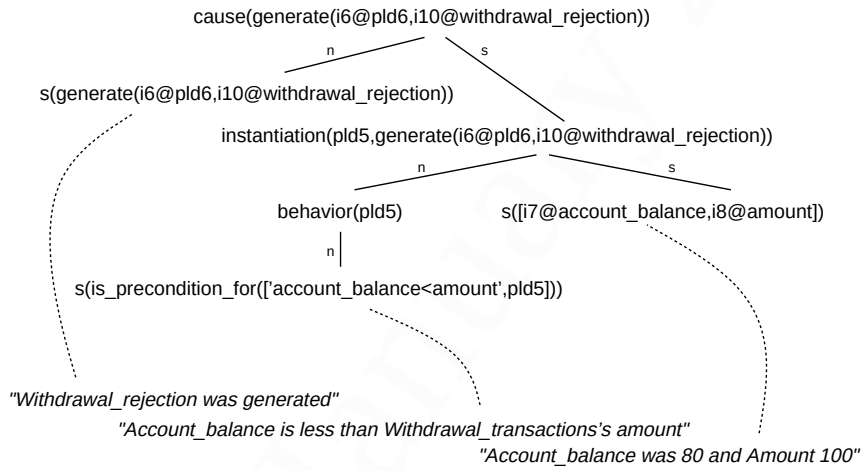


Fig. 6.25. Deep explanation for the question *"Why was my withdrawal rejected?"*

It should be noted, however, that the deep explanation in Fig. 6.25 is not shown in all its complexities. Instead of the characterizations used as

leaf nodes in the explanation structure, there should be full attribute-value structures that specify the linguistic realization of the content elements. If this full deep explanation were sent to a surface generator, we could get the realization below.

system: `Amount_rejection` was generated because `account_balance` is less than `withdrawal_transaction`'s `amount`. `Account_balance` was 80 and `Amount` 100.

In this text, the word *"because"* corresponds to the plan operator `cause`, and the content elements (i.e. the leaf nodes) are realized as clauses in natural language.

Generating an Input Justification. Since the customer's original withdrawal was rejected, the system will request a smaller amount to be withdrawn. This corresponds to flow `New_withdrawal` and receive construct `pld7` in Fig. 6.23.

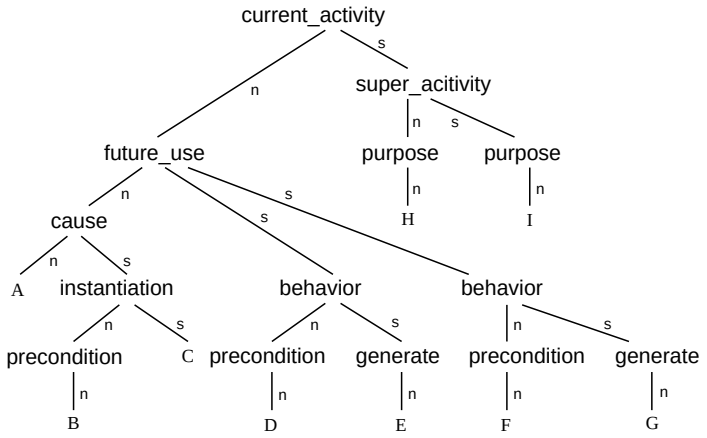
user: Why do you need `New_withdrawal`?

system: I'm verifying the amount as part of the processing of transactions. `New_withdrawal` was requested because `account_balance` was less than `withdrawal transaction`'s `amount`. `Account_balance` was 80 and `amount` 100. If `New_amount` $\leq$ `account_balance` and `New_amount` > 0, then `Ok_withdrawal` is generated. If `New_amount` $>$ `account_balance` or `New_amount` ≤ 0, then `Aborted_withdrawal` is generated.

In this case, the planning was a bit more complicated, and all the operators in Fig. 6.2 and Table 6.3 were needed. The operator structure of the generated deep explanation is given in Fig. 6.26, together with the linguistic realization of the isolated content elements. When coordinating the elements to form the final explanation above, though, the surface generator may have to modify these proposed grammatical and lexical realizations.

Related Work. Combining executable modeling languages, tracing mechanisms, and explanation generation techniques yields a novel approach to comprehension in CASE environments. There are no extant environments offering this combined approach to model validation, although the same techniques in isolation have been experimented with in many kinds of environments.

The idea of using executable specifications for validation purposes has been known for several years, but is not widely used in current practice. Our explanation component builds on many of the same basic principles for model transformation and execution as found in other environements. Some of these also provide advanced control of the execution, such as step-by-step execution, breakpoints, and display of active elements and events either graphically or textually, e.g. [164] and [241]. Although database approaches to tracing are difficult to find in environments for conceptual modelling, many of the principles employed in our tracing component are similar to those found in some programming environments.



element	*possible realizations of content elements*
A	*"New_withdrawal was requested"*
B	*"Account_balance was less than withdrawal transaction's amount"*
C	*"Account_balance was 80 and withdrawal transactions's amount was 100"*
D	*"New_amount <= account_balance and new_amount > 0"*
E	*"Sending of Ok_withdrawal is generated"*
F	*"New_amount > account_balance or new_amount < 0"*
G	*"Sending of Aborted_withdrawal is generated"*
H	*"I'm verifying the amount"*
I	*"I'm processing transactions"*

Fig. 6.26. Operator sturcture of deep explanation for the question *"Why do you need* `New_amount` *?"*

In Snodgrass' software monitoring system [347], traces are stored as temporal relations for recording dynamic information like call sequences. A temporal query language TQuel can then be used to retrieve the recorded information. In LeDoux's YODA system [226], the execution history of concurrent Ada programs is viewed as a stream of events. Recorded events are those deemed useful for program debugging, and include information about executing tasks and reads and updates of selected (simple) variables. A query processor interpretes queries written in PROLOG, which can have some temporal operators included.

Whereas Snodgrass' software monitor does not trace static objects, the changes made to state components is a primary concern for our purposes. Additionally, our general trace schema includes justifications of occurring events. The YODA system is solely focusing on debugging of Ada programs, whereas our schema provide more general representations which can be used for different languages, as exemplified by PPM and PLD. Our query language

does not provide the same temporal expressiveness , however it provides high level queries tailored for the purpose of explanation of model behaviour. Also, as noted above, our tracing component makes use of model knowledge, for instance knowledge about hierarchical relationships among processes and among PLD constructs.

CASE environments with extensive facilities for explaining the execution of conceptual models are not available today. A component resembling our explanation component, though, is included in the Gist environment [359]. Gist is an executable textual representation language, and having symbolically executed a Gist model, the system can translate the recorded trace into natural language. But this component is a pure paraphraser — there is no user-tailoring or context-tailoring — and the paraphrase's focus and structure is fixed since the user cannot ask specific questions to the generation component.

In Kalita's system [198], which is a report system, a component for reporting on the status of executing models is used. The model itself is represented as a hierarchical Petri net with rules governing the transitions between states. As opposed to the paraphraser in Gist, Kalita's paraphraser can be activated throughout the whole execution of the model. However, it is not possible to vary the generated text according to user, context, or question in his system, either.

In the field of expert systems, though, we can find some explanation generation components resting on the same general approach as ours (for example XPLAIN [360] and EES [295]). The two explanations generated above correspond to their *"why"* and *"how"* questions, and in EES there are plan operators building up the same kind of explanation structures as we do. Still, in these systems the source models include rule bases that are considerably simpler conceptually than our executable models, and these rule representations are inherently very close to the desired explanation structures. So, even though their approaches have given rise to impressive results in expert systems, the explanation generation components are not directly applicable in the domain of conceptual modeling.

6.6.8 Advantages of the Integrated Approach

The comprehesion techniques supported by PPP can be used independently of each other. For instance, we often see model execution used as a standalone technique. We assume that an integrated approach as presented above increase the potential of validating conceptual models. We have particularly focused on the combination of model execution and explanation. This combination gives better means to understand the relationships between a conceptual model and its behavior as observed during execution. A user can normally only observe the outputs given when a model is executed, and give inputs as requested. By combining execution with explanation, the user can get a rationale both for requested inputs and computed outputs. This can

also have positive consequences for later projects, since it increases the users understanding of the modeling language.

Complexity reducing techniques suppress details as necessary for better communication with users. This technique can be exploited by explanation generation, and make it possible to provide multimedia explanations.

The explanation generator glues together the techniques to form one comprehensive technique. Explanation strategies encode knowledge of how to explain events occurring during model executions, and the final explanation includes references to both the conceptual model and the actual execution. The result is a multimedia explanation which enhances user communication and provides useful feedback to the modeling process. We will return to the practical use of the techniques in the last chapter of the book.

6.7 Chapter Summary

Pragmatic quality is the correspondence between the externalized model and the audience's interpretation of it. The framework contains one pragmatic goal, namely **comprehension**. For a large model, it is unrealistic to assume that each audience member will be able to comprehend all the statements in the model which are relevant to them. Thus, comprehension as defined above is an ideal goal, just like validity and completeness, and can often not be achieved. From the technical actors' point of view, that a model is understood means that all statements that are relevant to the technical actor to be able to perform code generation, simulation, etc. are comprehended by this actor.

We have in this chapter presented several activities to achieve pragmatic quality in more detail:

- Transformations: Generally to transform a model into another model in the same language. We have here focused in particular on filtering, concentrating on and illuminating specific parts of a model.
- Translation: A translation can generally be described as a mapping from a model in one language to a model containing all or some of the same statements in another language. The main uses of translation mechanisms presented in detail in this chapter are:
  - Model execution and prototyping to be able to check the dynamic behavior of the model.
  - Explanation generation to answer questions about a model and its behavior.

7. Means for Achieving Social Quality

We have increased the font size in Fig. 7.1 of the relationship of the quality framework that is looked into in this chapter.

The main goal defined for social quality is *agreement*. In Chap. 3 six kinds of agreement was identified.

- Relative agreement in interpretation: all I_i are consistent,
- Absolute agreement in interpretation: all I_i are equal,
- Relative agreement in knowledge: all K_i are consistent,
- Absolute agreement in knowledge: all K_i are equal,
- Relative agreement in model: all M_i are consistent,
- Absolute agreement in model: all M_i are equal,

Relative agreement means that the various projections or models are consistent. Absolute agreement, on the other hand, means that all projections are the same.

Tool support in this respect is most easy to device on achieving agreement in models created based on the internal reality of the stakeholders that are to agree. One can also support the specific process of achieving feasible agreement. Based on this, main activities for achieving feasible agreement are model integration with specific emphasis on conflict resolution in the integrated models.

7.1 Tool support for model integration

The general process has many similarities with view integration, which has been a topic of much research in the database community. The process can be considered as consisting of four subprocesses [123].

- Pre-integration: When more than two models are used as input to the process, one must decide on how many models should be considered at a time. A number of strategies have been developed such as [105]: binary ladder integration, N-ary integration, balanced binary strategy, and mixed strategies.
- Viewpoint comparison: Includes identifying correspondences and detecting conflicts among the viewpoints. Some types of conflict that may be detected are [105]:

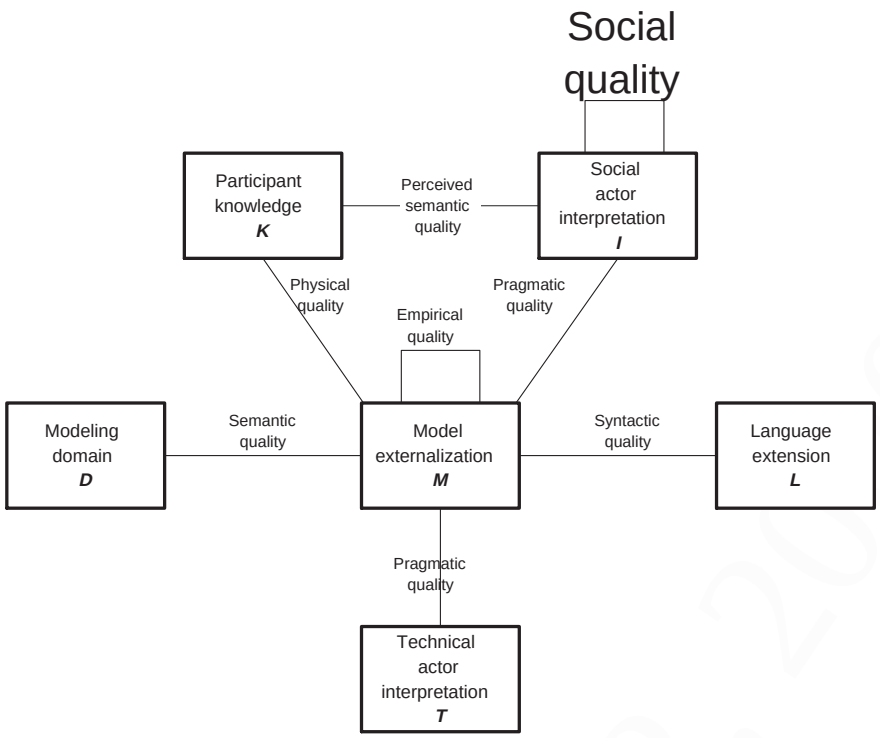


Fig. 7.1. Coverage of this chapter

  - Naming conflicts: Problems based on the use of synonyms and homonyms.
  - Type conflicts: That the same statements are represented by different symbols in different models.
  - Value conflicts: An attribute has different domains in two models .
  - Constraint conflicts: Two models represent different constraints on the same phenomena.
- Viewpoint conforming: Aims at solving the previously detected conflicts. Representations of statements in two different models can be classified as follows [123]: Identical, equivalent, compatible, and inconsistent. To deal with such conflicts traditional approaches are mostly based on either transformational equivalence or they entrust the skill of the participants by providing only examples valid for the particular model. According to [123] few approaches deal with inconsistent statements. A notable exception is Leite and Freeman [229]. They describes a way of dealing with conflicting rules in the modeling process. Rules are described in the rule-language VPWI where both data, actor, and process perspectives can be represented. A view consist of a set of rules. Mechanisms are provided to compare two different views of a given situation in order to identify, classify, and evaluate

discrepancies between the views, and integrate the solution into a single representation as illustrated in Fig. 7.2. A general strategy for viewpoint analysis is shown in Fig. 7.3. Here A and B, both system analysts, perform a modeling task. They both use the VPWI language to express their perception of the universe of discourse. They use different perspectives (process, data, and actor) and different abstraction mechanisms (generalization, aggregation) to improve their own view. Once a series of critiques are provided, each analyst alone solves the internal conflict and integrates their final perception into a view. After that, both views are compared and analyzed.

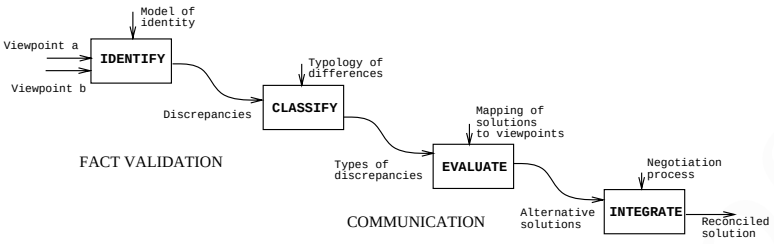


Fig. 7.2. Viewpoint resolution

Other useful techniques in this respect is the use of argumentation systems [70, 157] for supporting the argumentation process. These uses the IBIS [1]-approach proposed by Rittel [314] or extensions of this. IBIS focuses on the articulation of the key *issues* in the problem area. Each issue may have many *positions*, which are statements or assertions which resolve the issue. Each of the issue's positions in turn may have one or more *arguments* which either support or object to the position. Going from one node-type to another is done through so-called rhetorical moves. A more detailed overview of the types of such moves between nodes is given in Fig. 7.4, taken from [70]. E.g. an issue can be a generalization or a specialization of another issue, question an issue, a position or an argument, replace or be suggested by another issue or be suggested by a position.
Later work e.g.[60, 311] have extended this model. We will return with an example of the use of this in the end of this chapter.

– Merging and restructuring: The different models are merged into a joint model and then restructured. The latter involves checking the resulting model against criteria for semantic, pragmatic, and social quality.

Generally, it is not to be expected that matching apart from syntactic matching can be performed totally automatic. Model merging can be supported in several ways, having computerized support for manual integration, possibly with the use of CSCW-techniques.

[1] Issue Based Information Systems

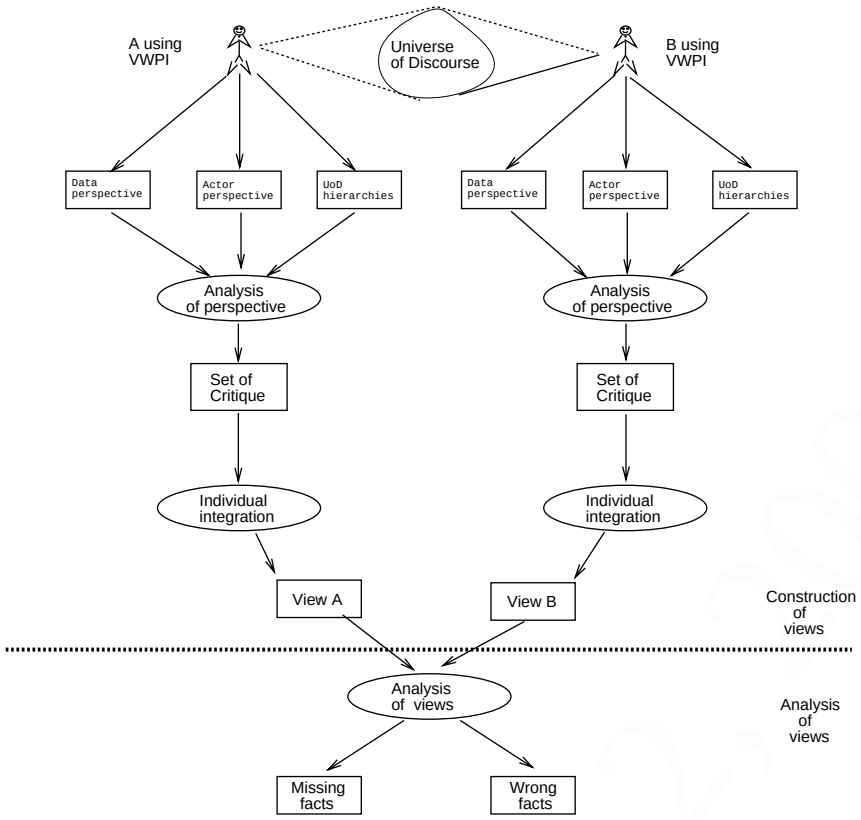


Fig. 7.3. Strategy for viewpoint analysis

– Computerized support of manual integration: Manual merging may be supported in various ways by exploiting modern user-interface technology. Working styles such as virtual paper, clipboards, cut and paste, and active structures can be supported.
  It can be useful to provide facilities to track the transformations performed on a predecessor model, i.e. recording modeling history during updating and filtering. The changes can be recorded textually, or shown explicitly relative to the diagrams of the predecessor model by using special notation in the diagram. In the latter case, cut and paste facilities across windows between models, greatly improve the merging process.
– Automatic integration support: The result of automatic merges are useful only in some cases e.g. if all components of the different models have got unique identifiers. However, the use of formal modeling techniques opens up for more extensive integration techniques where for example structural conflicts may be resolved. This is useful in most modeling situations where different participants may have different perceptions on the area and a

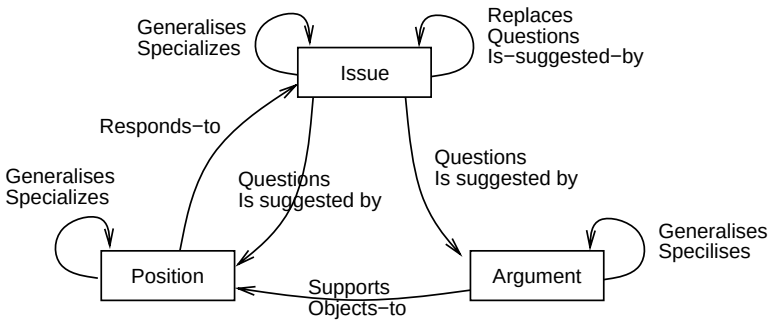


Fig. 7.4. Relationships in IBIS

possible CIS-implementation to address the perceived problems. To address such conflicts, a number of conflict resolution techniques are proposed, see e.g. [188, 353].

- CSCW support: CSCW techniques can be applied to create new arenas for dialogue. In addition to face-to-face meetings, the integration effort can take place by applying more advanced workstation and networking technology in cooperative sessions. The negotiation session may be synchronous or asynchronous [103].
  - Synchronous negotiations: In the synchronous case all candidate models are available to all involved participants in the session through a shared workspace for comments and comparisons. Several modes of working are supported [180]:
    - The public screen and public desktop allow all participants involved to view one participants screen or physical desktop. This mode implements multi-casting of the changes made to one screen or desktop. Only the owner of the screen/desktop may update it.
    - Desktop on screen and desktop on desktop allow participants to draw sketches and comment the contents of other developer's screens or desktops, using overlays. Experimental evidence indicates that users easily differentiates up to three overlays.
    - The shared tool mode allows all participants to simultaneously view and edit conceptual models.

    The usefulness of these modes are dependent on the presence of synchronized sound or video and the possibility of flexible switching between modes.
  - Asynchronous negotiations: In this case one relies on written communication in the form of comments and cut-and-paste versions, and multimedia annotations to the artifacts. The multi-media messages passed between participants are based on the candidate versions. The messages can contain annotations to one or several candidate versions in the form of synchronized pointing, drawing, writing, and speaking [124]. To annotate a local version of a model, one would add a transparent annotation

layer to it, where the annotation is entered. This may be played back in synchronous mode, playing back voice comments, the cursor movements of a stylus, as well as hand-drawn and typed messages.

Below, we present an outline of an approach to model-integration in PPP.

7.2 Model Integration in PPP

Model integration is a general process where one based on several conceptual models create a single model.

Model integration techniques can be useful on many different areas of conceptual modeling, not only to enhance social quality of models. We have divided the overview given here in three areas:

- Intra-project integration.
- Inter-project integration.
- Inter-organizational integration.

7.2.1 Intra-project Model Integration

In this case, the model integration happens within a project, integrating models that are created specifically in the project. All the actors that are the sources of the models to be integrated are expected to be available. The languages that are used for modeling in this case are all part of the PPP conceptual framework presented in Chap. 2.5.

Several cases can be perceived:

- A set of models created by the same actor in different modeling languages need to be integrated. This typically needs to be done before integrating the views of the actor with models based on the local reality of other actors. To support this one can use the same links between models that are utilized for the driving-questions techniques discussed in Chap. 5. In addition, one should try to get the vocabulary consistent, but as part of building up the thesaurus also indicate the semantic distance between similar and dissimilar terms [199].
- Two or more models developed independently by different actors in the audience are to be integrated. Depending on the overlap of the areas of modeling, one should expect more or less overlap in the models. Generally, one should in this case rather look after similarities than differences in the models to be compared. When the models are written in the same language, and using the same parts of the language, one can compare the constructs in the models, and in addition apply the thesaurus developed to discover potential synonyms. An example of this kind of matching is given in Chap. 8. As indicated in work on model integration of structural models [123], the same situation can often be described using different

modeling constructs. Because of this it might also be beneficial to compare the statements of the models and not the modeling constructs directly, but also here use fuzzy search applying the thesaurus information.
If the models are developed using different parts of the language, the comparison is limited to the parts that are used in both models, when comparing at the component-level. When comparing at the statement-level, there is no change.
If the models to be compared are developed in different languages, e.g. PPM and ONER, one can use the driving questions technique presented in Sect. 5.3 to create preliminary models in the other languages, and use these for the first comparison. Alternatively, one might compare on the statement-level directly also here.

– The models have a common predecessor model which they are based on. The models to be integrated can either be created by the same actor or by different actors. This area include the traditional merging of two variant models as discussed by Andersen [7], or the integration of viewspecs as discussed by Seltveit [331]. In both cases, one start out with a set of statements in the original model, and based on this adds and/or deletes a set of statements. When using a filter, one starts out deleting a set of statements from the original model, which are to be put back at a later point in time. The main approach in this situation can thus apply the explicit knowledge of the statements that are inserted or deleted as part of modeling.

An example of the conflict resolution process within a project using techniques such as rule-hierarchies, IBIS and CATWOE are presented briefly below. In the example, the main participants were AS, OIL, JK in addition to the group developing the new system, denoted 'GR' below.

As a starting point for this modeling, the root-definitions [61] for all participants were established using the CATWOE-technique.

This basically answers the following question:

> **Who** is doing **what** for **whom**, and to whom are they **answerable**, what **assumptions** are being made, and in what **environment** is this happening?

Costumer is the 'whom', **A**ctor is 'who', **T**ransformation is 'what', **W**eltanschauung is 'assumptions', **O**wner is the 'answerable', and **E**nvironment is the environment.

Starting out with the CATWOE-analysis is deemed important to enable the construction of individual models for the participants. The questions used in the CATWOE-analysis are similar to what Gause and Weinberg terms 'context-free questions' [132], which they also propose to ask the stakeholders of a project very early in development.

As an example, OIL gave the following response:

– Customer (the whom): The organizing chair of the conference and SEVU

– Actor (the who): GR and JK.
– Transformation (the what): Good communication and coordination between SEVU and the organizing chair.
– Weltanschauung (the assumptions): A well-organized conference is of major importance to the research group.
– Owner (the answerable): AS.
– Environment: The university.

As part of the CATWOE-analysis, the following goals were identified. The source of each goal are indicated in *italics*.

– **R1002**: "It is recommended for the group to get a good grade on the project" *GR*.
– **R1003**: "It is recommended for the group to make the customer satisfied" *GR*.
– **R1004**: "It is obligatory for the group to keep within the time-budget" *GR*.
– **R5**: "It is recommended for the organizing committee to create interest for the conference" *JK*.
– **R152**: "It is obligatory for the program coordinator to keep track of papers and reviews of papers" *JK*.
– **R902**: "It is obligatory for the organizing chair to have good cooperation and coordination with SEVU" *OIL*.
– **R4**: "It is obligatory for the organizing committee to create interest for the conference" *AS*.
– **R504**: "It is recommended for the conference system to support other conferences than ISDO95" *AS*.

This initial situation is depicted in Fig. 7.5. Rules and arguments are given in boxes, using arrows for relationships annotating these with **O**, **R**, **P**, **D**, **F**, and **REL** (for an undetermined relation). The sources of both rules and relationships are indicated in ellipses.

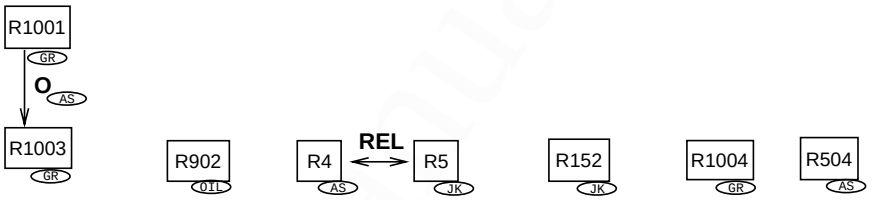


Fig. 7.5. Rules based on CATWOE analysis

Goals and rules that appear at this stage are not necessarily on a highest level, thus one can work upwards from the existing goals. Some rules identified on this basis which were generally applicable for the first modeling task were:

- **R3**: "It is recommended for a research group to arrange conferences". This goal recommends the goal **R504** according to *AS*, with the additional argument **A1**: "Many conferences of this kind are arranged. If we can come up with a general system that can be used in other conferences arranged in the community, this will be a contribution from us to the community".
- **R501**: "It is recommended for our research group to arrange this particular conference" *AS*. This is motivated both by rule **R3** and other goals not given here.
- **R401**: "It is recommended to apply the services of a local conference organization" *OIL*.

The overall goal-model can also be developed downwards. Below, we list some of the rules being pursued regarding the choice of the target platform for the CIS:

- **R505:** "It is obligatory that the conference system run on Unix" *JK*. This is recommended by **R152** and supported by the argument **A2**: "I (JK) will be the main user of the system, and I use Unix for other work in connection with the conference".
- **R506**: "It is recommended that the system run on PC using MS-Windows" *AS*. **R504** using argument **A3** "Many people use PC's" is regarded by *AS* to obligate **R506**. At the same time, this rule is discouraged by the combined effect of **R1004** and **R505** according to JK due to the limited resources of the project.

A goal-hierarchy only including the rules mentioned specifically in this chapter is depicted in Fig. 7.6.

Here an *issue* is identified: The platform for the conference system. This issue, with the identified *positions* and *arguments* for these can be transferred into an IBIS-like tool [70], where a concentrated argumentation process on this issue can be performed. Figure 7.7 gives an overview of the situation after additional positions and arguments have been added, some of which can be transferred back into the goal-hierarchy. The *decision* of the argumentation was to choose Unix for the first version of the system, but perform a generic design of the system to make later porting of the system easier. Since arguments in this situation can be further motivated by additional rules, one are also able to model the assumptions of arguments as suggested in the extension of IBIS presented in [311].

Although discrepancies are resolved, one do not totally discard the other alternatives. The pruned goal-hierarchy is showed in Fig. 7.8, also indicating that **R505** is linked to an issue.

In this way one may prevent premature closure by capturing the variety of views. If it appears later in the project that new factors e.g. technical constraints indicates that one should re-assess an issue, this can easily be done. If many different positions are present on an issue, this might indicate an area where changes are more likely to be done in the future. This might

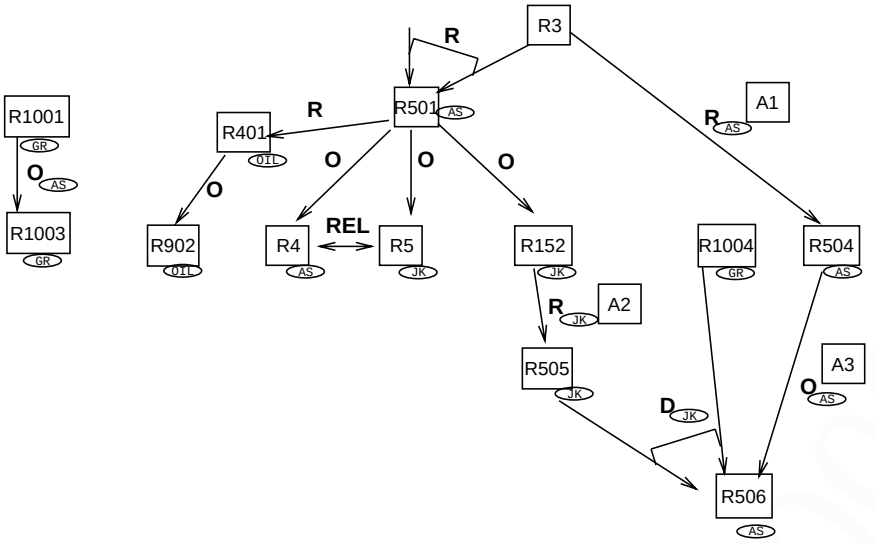


Fig. 7.6. Goal-hierarchy extended from CATWOE analysis

suggest that more effort should be used up front to prepare this area for further development at a later stage. It is also important to note the design decisions that are made based on the particular supporting software and hardware actors in the project, since it is likely that one might want to change this underlying architectural platform at a future time. Generally, the argumentation structure which is used is believed to be a good way to capture design decision, thus in a maintenance project enable a better understanding of why the existing application system was made in the way it was.

7.2.2 Inter-project Model Integration

The models are developed in two different projects within the same organization. Examples are:

- A replacement system is made, based on functionality from one or more previously independent application systems which have been developed and maintained using conceptual models. These models can be expressed in either the same or different languages.
- A maintenance project, further developing an application system which have been developed using conceptual models in the case where one do not automatically reuse the whole existing conceptual model. The existing models can also have been created through re-engineering.
- An enterprise-wide model has been developed, and areas subsumed by this model are to be developed in more detail in a development project. This is similar to the above situation.

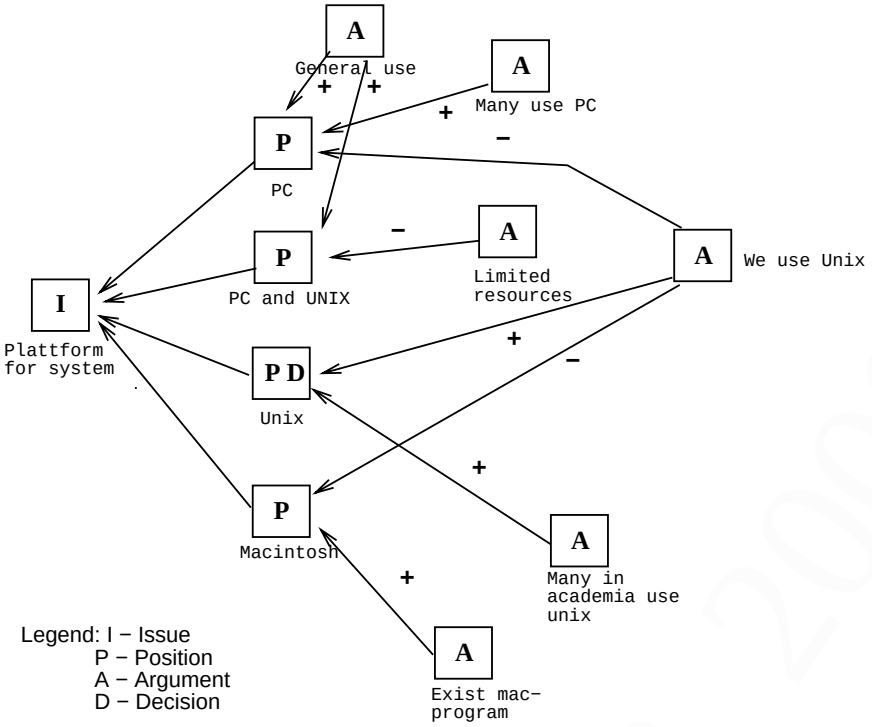


Fig. 7.7. IBIS-network on the issue of system platform

– Several application system needs to access similar data, and one needs to compare the structural models describing the data used by the different systems to ensure that they are consistent.

In this case, additional terminological problems may appear. In the Sweden Post case, a case study in the Tempora project, what the Post was selling was called a 'product' by the accounting department, and an 'article' by the marketing department. The data associated with 'article' and 'product' was different although both refer to the same phenomena [331]. To address problems of this sort, a two-level thesaurus-structure, including also a taxonomy of application domains [138] might be useful.

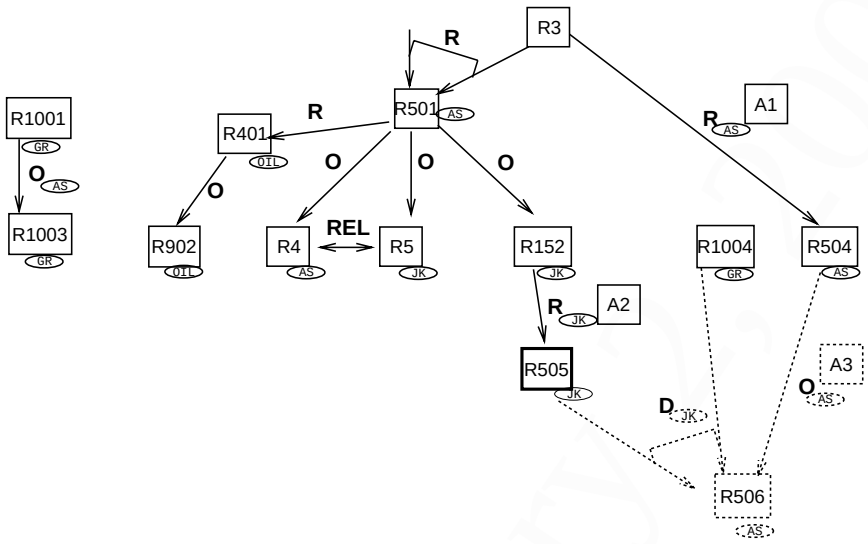


Fig. 7.8. Pruned goal-hierarchy after argumentation process

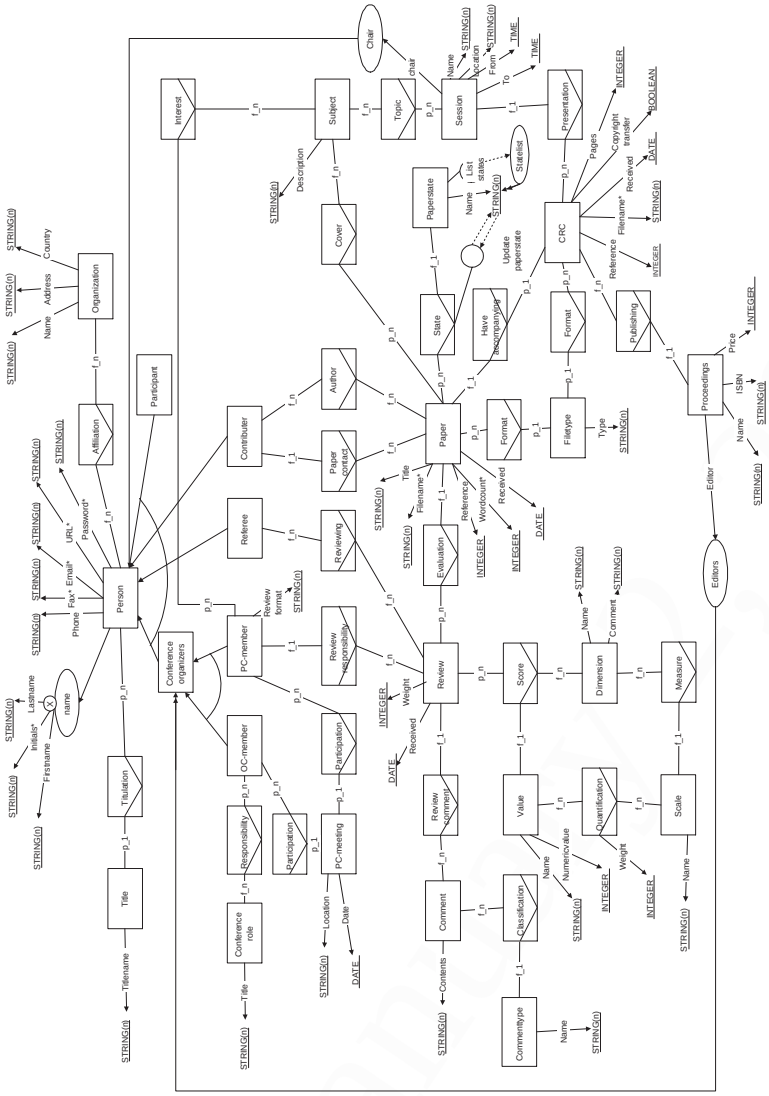


Fig. 7.9. ONER-model for the IFIP-case

To further illustrate potential problems in this area, we have compared the ONER-model supporting the paper-process both from the conference case (in Fig. 7.9) and as it is modeled by Yang [404] based on the original case-description for an IFIP-conference (in Fig. 7.10), i.e. to simulate the comparison of an earlier application system supporting an area and the needs for a new one. The model of the IFIP-case also consisted of a separate scenario in connection with the practical arrangement of the conference, which does not overlap with our model at all.

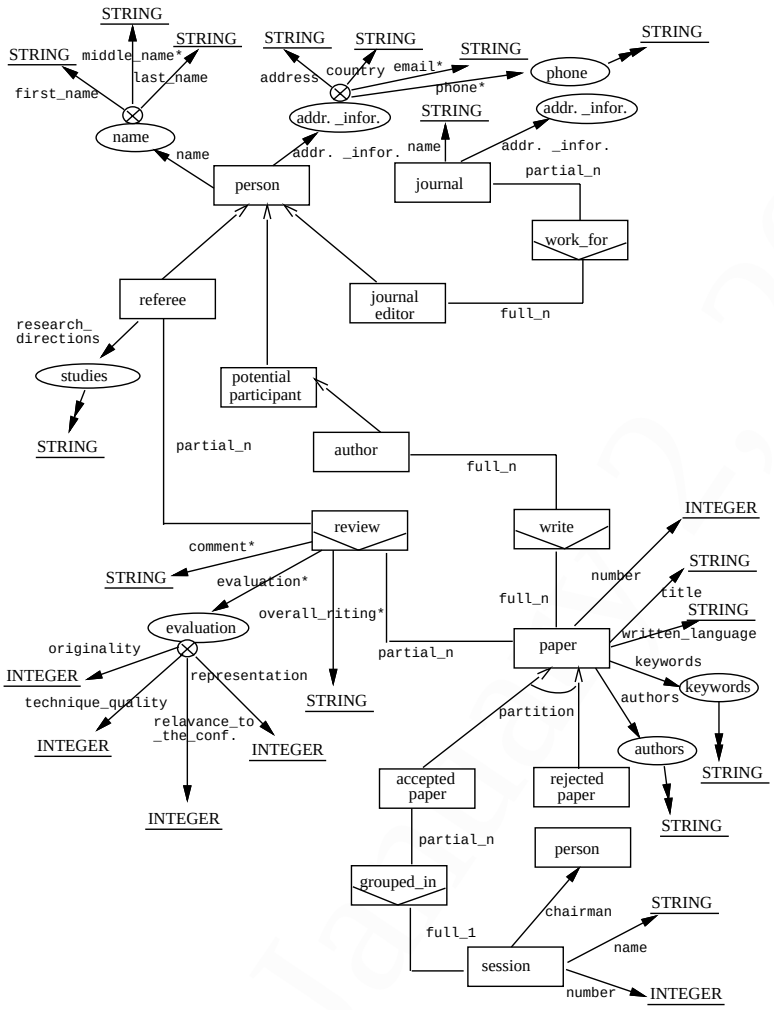


Fig. 7.10. ONER-model for the traditional IFIP-case (From [404])

Comparing these models, we observe the following:

- There are some common entity-classes, e.g. person, referee, paper, and session.
- There are some common types, e.g. name, address, country, email, phone, title (of paper), name (of session). The specification of name is somewhat different, and country and address is only indirectly linked to the person in our case, through the affiliation relationship, and not directly as in the IFIP-model.
- There are entity-classes that are given synonymous labels. (participant/potential participant).
- There are entity-classes in the old diagram that are represented as relationship-classes in the new diagram, e.g. review.
- There are similar entity-classes that are represented with different labels e.g. CRC vs. accepted paper.
- There are similar relationship-classes that are represented with different labels, partly because the old model use role-names on relationship-classes and the new use relationship-names e.g. Presentation vs. grouped_in, author vs. write.
- There are aspects that are represented as sub-classes in the old version (accepted/rejected paper) which are represented as values of an attribute in the new (state.name).
- There is a type in the old diagram that is represented as a scenario in the new diagram, including values of attributes. (evaluation vs. dimension/value/scale.) The large difference here is caused by different requirements to flexibility of the solution.
- There are attributes in the old diagram that are represented as entity-classes in the new e.g. comment.
- There are similar attributes with different labels e.g. number vs. reference for paper.
- There are types in the old diagram that are represented as an entity-class and a relationship-class in the new diagram (Paper cover subject vs. paper.keywords and PC-member interest subject vs. referee studies).

In addition to the above differences, there are areas which are included in the old diagram that are not included in the new, and vice versa.

7.2.3 Inter-organizational Model Integration

This can be interesting in several cases. A model is developed in an external organization (e.g. a model of an application framework), and one wants to find out to what extent this model is sufficiently close to a model developed within the organization for the possible adoption of the framework. The models to be compared are not necessarily written in the same language. Another possible scenario for the need of performing inter-organizational model integration is the case where several organizations needs to coordinate parts of its data-processing, for instance when applying EDI, or when temporarily cooperating

to form a virtual organization. Obviously also more permanent cooperation through e.g. mergers can provoke the need for model integration of this sort.

Based on the above we would similarly expect large differences between models in inter-organizational model integration also when they are expressed in the same language. When being expressed in different languages the problems would potentially be larger, even if the languages share the same perspective.

Although matching can be done on type e.g. matching two processes based on that they receive the same type of input and return the same type of output [297], it is often dependent on using information in the labels of the model, since there is a large number of e.g. process-diagrams that have the same port-structures and that diagrams with different port-structures might contain similar functionality. Thus even if the use of analogy for general reuse [58, 251] can be a powerful mechanism, one can not expect to be able to use this in all cases.

7.2.4 Outline of an Approach to Model Integration

One way of attacking this area within PPP is to use a (technical) actor model as an outset for the comparisons on a high level. Features of the actor model that can be utilized for model integration purposes are:

- Data accessible by the different actors.
- The capabilities of the actors. This might both be existing capabilities and potential capabilities. The capabilities of an actor in certain roles can also be interesting.
- The type of the items that the actors might receive and send to be able to utilize the capabilities of the actors. This includes descriptions of the interface of the actor.
- The rules that applies to the actor.
- Actual and potential supporting actors of the actor.
- Different abstractions of the actor such as ONER-models and PPMs.

We have opened up for the support of matching models written in different languages, and possibly using different perspectives. To be meaningful, there must be some potential for overlap between the models of the different languages. Some examples are the comparison between an ER-model and a ONER model, where the ONER language subsume the ER-language. This can be supported by having translation facilities between the modeling languages, and perform comparisons based on transformed models [12]. More complicated is the problem of comparing say a process and a data model. As we have illustrated, there are some connection between the data and process modeling languages in the conceptual framework, thus, these can be utilized for a preliminary matching using the same principles that are used in the technique of driving questions. Another approach to this, is the use of a common underlying model, as is done in e.g. ARIES [24].

The actor-modeling language is meant to both give a view independent of traditional modeling languages, by looking upon an actor as having certain capabilities, and investigating actor support relationship between technical actors. It also cover the use of traditional data, process, and rule-modeling. It necessitates a translation from other models to this model to be useful in extended inter-organizational reuse and matching. In addition to the use of the actor-model, it should be possible to use the translations of conceptual models into statements directly.

In cases where the models to be integrated do not have a common predecessor, one should as a rule try to find the potential overlap between models and not the difference, since at least in our experience the difference often seems to be larger than the overlap. When having found models with certain overlap, one might apply a transformational approach to matching, thus finding how many inserts and deletes of statements that must be done to transfer one model to another, by utilizing the symmetry between inserts and deletes. For models in specific languages e.g. ONER-models, one should investigate heuristics in this search to make it possible to apply the technique more efficiently. Specific approaches for merging models that have the same predecessor have been discussed in [7, 331] and was discussed above. We will also look upon this in some more detail in the next chapter.

7.3 Chapter Summary

In this chapter we looked upon what we have termed social quality. The main goal defined for social quality is *agreement*. Six kinds of agreement was identified.

- Relative agreement in interpretation
- Absolute agreement in interpretation
- Relative agreement in knowledge
- Absolute agreement in knowledge
- Relative agreement in model
- Absolute agreement in model

Main activities for achieving feasible agreement are model integration and conflict resolution. The general process has many similarities with view integration, which has been a topic of much research in the database community. The process can be considered as consisting of four subprocesses.

- Pre-integration
- Viewpoint comparison
- Viewpoint conforming
- Merging and restructuring

Model merging can be supported in several ways, having computerized support for manual integration, possibly with the use of CSCW-techniques as one avenue for exploration.

Model integration techniques can be useful on many different areas of conceptual modeling, not only to enhance social quality of models. We have identified three areas:

- Intra-project integration. Model integration is performed within a project, integrating models that are created specifically for the project
- Inter-project integration. The models to be integrated are developed in two different projects within the same organization.
- Inter-organizational integration. Models developed in different organizations are integrated.

The briefness of this chapter partly mirrors the fact that comparatively little work has been done on the area outside the database community. Looking into proceedings from current conferences though, we see that much work is currently done in the area, and we would expect many interesting techniques and tools to surface in the years to come.

8. A Methodology for Conceptual Modeling

Whereas the last four chapters have presented different techniques for conceptual modeling, we will in this chapter present a more detailed way of applying a conceptual framework (in this case PPP) as part of a development methodology. This include process heuristics, use of modeling in development, use, and maintenance of computerized information systems, and configuration management of conceptual models. In the next section, we will put this approach into a wider context by presenting a classification framework for computerized information systems support.

It is important to appreciate that what is presented in this chapter is only one of several possible approaches to conceptual modeling in computerized information systems support in organizations. It should not be looked upon as a suggestion for an ultimate methodology, but rather as a set of possible guidelines. A comprehensive CIS-development and maintenance methodology would typically take up many hundred pages of text, thus not being possible to include in a book of this kind in its entirety.

We will base much of the treatment in this chapter on the case study described briefly in the Chap. 1, describing an information system for supporting the arrangement of a scientific (IFIP) conference.

IFIP is the acronym for International Federation for Information Processing. An IFIP working conference is an international conference that provides an opportunity for the computer scientists from IFIP member countries to discuss and interchange research results and new ideas on selected research fields.

The management of such a conference is usually done by two cooperating committees. The *program committee* (PC) handles the contents of the conference, say, the reviewing of papers, comprising sessions and tutorials, etc. The *organizing committee* (OC) handles the administration work, e.g. sending out invitations, registration of attendants, arranging time and places for sessions, dealing with financial matters, etc.

The system developed in the case was to support the work of the program committee and the program organizers. The case will be further described throughout the chapter. In addition to using the case to illustrate the methodology, we also use it to give examples of the different sets that are described in connection to the quality framework.

Before going into the detailed description of the methodology, we will present a classification framework for methodologies for computerized information systems.

8.1 Classification of Methodologies for Computerized Information Systems Support

When deciding on relevant dimensions for a classification framework, we have asked the questions why, when, what, how, who, where, and for how long in the context of CIS-support. We have classified methodologies according to the following areas:

- Why do we attack the overall problem area in the way we do? This is covered by the "Weltanschauung", i.e. the underlying philosophical view of the methodology.
- When is the methodology applied? We have termed this aspect coverage in process, meaning the main tasks that are covered by the methodology.
- What part of the CIS-portfolio of information systems is supported by the methodology? we have termed this aspect coverage in product.
- How do the methodology help in achieving the goals of CIS-support? We have specifically concentrated on reuse and representation of product and process in the methodology, with emphasis on conceptual modeling.
- Who is involved and where are the changes done? This is discussed under the area of stakeholder participation.
- For how long has the methodology been used. We term this aspect maturity: Is the methodology mature, being used for a long time in many organizations, with tool-support and support for evolution of the methodology.

Below, we will define and discuss each area in more detail.

8.1.1 "Weltanschauung"

FRISCO [109] differentiate between three different views of the world:

- Objectivistic: "Reality" exists independently of any observer and merely needs to be mapped to an adequate description. For the objectivist, the relationship between reality and models thereof is trivial or obvious.
- Constructivistic: "Reality" exists independently of any observer, but what each person possess is a restricted mental model only. For the constructivist, the relationship between "reality" and models of this reality are subject to negotiations among the community of observers and may be adapted from time to time.

– Mentalistic: To talk about "reality" as such does not make sense because we can only form mental constructions of our perceptions. For the mentalist, what people usually call "reality" as well as its relationship to any model of it is totally dependent on the observer.

Methodologies can be categorized as being either objectivistic or constructivistic. The "weltanschauung" of a methodology is often not explicitly stated in descriptions of the methodology, but often appears only indirectly. Since different underlying philosophies may lead to radically different approaches, it is important to establish this. The distinction into objectivistic and constructivistic is parallel to the distinction between objectivistic and subjectivistic in the overview of Hirschheim and Klein [170]. Hirschheim and Klein also distinguish along the order-conflict dimension. In this dimension, the order or integration view emphasizes a social world characterized by order, stability, integration, consensus, and functional coordination. The conflict or coercion view stresses change, conflict, disintegration, and coercion. These two dimensions were originally identified by Burrel and Morgan [50] in the context of organizational and social research.

Based on the discussion in the introduction of the book, it should come as no surprise that we find it beneficial to adapt a constructivistic world-view. Note however that we have a somewhat different approach to constructivism than the one described in the FRISCO-report.

8.1.2 Coverage in process

Do the methodology address:

– Planning changes to the overall CIS-support.
– Development of application systems.
– Use and operation of application systems.
– Maintenance and evolution of application systems.
– Management of planning, development, operations, and/or maintenance of application systems.

One or more of the above areas can be covered, more or less completely and in varying degrees of detail. More detailed specifications of dimensions of development methodologies are given by Blum [29], Davis [79] and Lyytinen [248]. Whereas Davis classifies a methodology according to the way it is able to address varying user-needs over time, Blum classifies development methodologies in two dimensions; if they are product or problem-oriented, and if they are conceptual or formal. The product vs. problem-oriented dimension as discussed by Blum is in our view a distinction on the *part* of development that is covered (analysis or design). The conceptual vs. formal distinction is covered under representation of process and product below. Lyytinen includes aspect covered by the "weltanschauung" and representation of product and process,

in addition to linking technical, linguistic, and organizational aspects in a development methodology.

We claim that a comprehensive methodology should cover both development, maintenance, use, operations, planning and management in an integrated manner. The emphasis in this book is put on development and maintenance, but also the usage aspect is important, enabling the different end-users to make sense of the existing applications system in the organization, both to be able to use them more efficiently, and to be able to come up with constructive change-request and ideas for more revolutionary changes in the CIS support of the organization when the environment of the organization is changing. We also claim that it is beneficial to not differentiate between development and maintenance in most cases. This is based on the figures appearing in our survey-investigation and in accompanying work presented in [171, 207].

It is both natural and desirable for CISs to change. As shown both in our own and other surveys, approximately half of the work which is normally termed maintenance is in fact further development of the information systems portfolio, and should be given credit as such. On the other hand, almost half of the new systems being developed are replacement systems, being functional maintenance not enhancing the functional coverage of the portfolio or in other words, what the users can do with the systems. Thus seen from the end-users point of view, a better assessment of information system support efficiency seems to be found by blurring the old temporal distinction between maintenance and development, and instead focus on the percentage of functional development. This is difficult to achieve when having a large mental and organizational gap between development and maintenance, even though the actual tasks being done have many similarities. Maintenance has traditionally been looked upon as a more boring and less challenging task than development [137]. Even if there are signs on that this view might be changing [224] this still appears to be the prominent view among practitioners.

Swanson [357] recognizes the similarities of the tasks of development and maintenance, but still argues for keeping the old distinction based on the following perceived differences:

– As also noted by Glass [137], a large proportion of traditional maintenance work is to perform "un-design" of existing systems, finding out what the system does. We argue that with modern development approaches where as much as possible of the work should take place on a specification and design level, the difference will be smaller. We also note that because of the large amount of replacement work of often poorly documented application systems, code understanding problems are often just as important when developing "new" systems as when maintaining old systems today. Code and design understanding will also often be a problem when reusing components and other artifacts from other projects, and during traditional

development, when due to changing work load, developers have to work on other peoples development deliverables for instance during system-test, or because developers are transferred to other projects.

– It is generally believed that "Maintenance of systems is characterized by problems of unpredictable urgency and significant consequent fire-fighting. In difference to new systems development, which is buffered from the day to day tasks of the users, the systems in production is much more visible" [357]. Traditionally, it has been found that approximately 20% of the maintenance work is corrective maintenance [234], and our own result of 26% [215] and 21of work that is performed to do immediately necessary corrective maintenance on the application level, we found in our own investigation [215] a percentage of 6%, the similar figure in Lientz/Swanson [234] being 12%. The total amount on corrective maintenance on the individual systems in our investigations was 15%. Jørgensen [194] indicate that the assessed corrective percentage of the work used on maintenance often might be exaggerated since these kind of problems are more visible for management. They found in their investigation of individual maintenance tasks that even if 38% of the changes were corrective, this took only up 9% of the time used for maintenance. Management assessed the percentage of corrective maintenance to be 19%. Those managers who based their answers on good data had a result of 9% corrective maintenance. Also in our investigation, we found a similar tendency, on the data of the maintenance task of the individual systems, those reporting to have good data, reported that only 8% of the work effort was corrective maintenance, 4% being emergency fixes. The same effect on over-assessing the amount of corrective maintenance has been reported earlier by Arnold [10]. In a case study on maintenance in American and Australian organizations of COBOL applications, corrective maintenance was reported to constitute a minor problem where less than 10% of the programs studied had undergone more than two corrective maintenance activities during their lifetime [382]. The significant factor were program complexity and programming style. Related to this is the results of a survey reported on in [87] which gave no conclusive evidence that organizations using modern development methods used less time on maintenance activities. On the other hand, time spent on emergency error corrections as well as the number of system failures decrease significantly with the use of modern development methods. Systems developed with modern methodologies seemed to facilitate making greater changes in functionality as the systems aged, and the request from users seemed more reasonable, based on a better understanding of the system.
The problem of many small maintenance tasks done more or less continuously seems to be increased by how maintenance is often done, in an event-driven manner. In the Jørgensen investigation [195], where 38% of the tasks were of an corrective nature, as much as 2/3 of the tasks where classified

to have high importance by the maintainers themselves. The problem of changing priorities as described by Dekleva [86] is closely related to this. Even if the problem of emergency fixes seems to be smaller than earlier perceived, a methodology uniting development and maintenance must take into account that one has to be able to perform rapid changes to software artifacts.

8.1.3 Coverage in product

Is the method concerned with the planning, development, operating, use, and/or maintenance of

- One single application system.
- A family of related application systems.
- The whole portfolio of application systems in an organization.
- The totality of goals, business process, people, and technology used within the organization.

Since the computerized information systems in the organization is also linked to the strategies, business processes, and people of the organization, a further natural extension is to integrate the planned change effort of all these areas, at least on a high level.

Over time, newer application systems originate in niches provided by older ones, and identifiable families of systems come to exist. Relationships among families are further established. In the long run, an organization is served more by its CISs as a whole than it is by the application systems taken individually. Based on this, we argue that it is beneficial for a methodology to make it possible to consider the whole portfolio and not only the single application system. For the end-users, it is not important which application system that is changed. What is important is that their perceived needs are supported by the complete portfolio. This do obviously not mean that one always need to consider the whole portfolio when enhancing the CIS-support of the organization.

Application systems are not developed in a vacuum. They are related to old systems, by inheriting data and functionality, and they are integrated to other systems by data, control, presentation philosophy, and workflow [369]. As reported in our investigations [171, 215], the most important reason for replacements apart from systems being unmaintainable, was integration and standardization of application systems. Often when doing this kind of integration, it can be useful to collect the functionality of several existing application systems into a new application system, something which is not well supported when having strict borders between what is inside and outside of an application system.

As noted in [357] the CISs of an organization tend to congregate and develop as families. By original design or not, they come to rely upon each other for their data. In Swanson/Beath [357] 56% of the systems where connected

to other systems through data integration. In our survey, we found that 73% of the main computerized information systems in the organizations surveyed were dependent on data produced by other systems. In 40% of the responses to this question *all* the main system which the organizations depend upon on a daily basis were regarded to be dependent on data produced by other systems.

8.1.4 Reuse of product and process

Reusing experience is a key ingredient to progress in any discipline. Without reuse everything must be re-learned and recreated; progress in an economical fashion is unlikely. Overviews of dimensions of reuse are given in [122, 216, 308], and we have based our overview and classification of this area on this work:

- By motivation: Why is reuse done. In addition to aspects such as productivity, other reasons for reusing existing solutions might be time to market, flexibility, evolvability, capacity, quality and management of uncertainty and risk.
- By substance: The essence of the items to be reused:
  - Artifacts reuse: The artifacts can for instance be code, conceptual models, designs, specifications, objects, components, text, architectures, estimating models or test data. In principle might all deliverables produced during a project be reused later in some way.
  - Process reuse: Formalizing and encapsulating software development procedure. Process reuse also means reusing skills and know-how, i.e. having a development and maintenance methodology can be looked upon as reuse in this sense.
- By development scope: The form and extent of reuse: This refers to whether the reusable entities are from a source external or internal to a project or organization. When being external to the project, one can also differentiate between vertical and horizontal reuse.
  - Vertical reuse is reuse within the same application area.
  - Horizontal reuse is reuse across application areas.
- By management mode: How reuse is conducted:
  - Planned reuse: The systematic and formal practice of reuse. Guidelines and procedures for reuse have been defined, and metrics are being collected to assess reuse performance.
  - Ad-hoc reuse: An informal practice, in which components are selected from general libraries.
- By technique: How reuse is implemented:
  - Compositional reuse is the use of existing artifacts as building blocks for new systems.
  - Generative reuse is reuse at the specification level by means of design and code-generators.

- By intention: Defines how elements will be reused:
  - As-is or black-box reuse is reuse without modifications.
  - Adapted. As with black-box, but with small changes due to porting issues resulting from a different technological infrastructure.
  - Modified or white-box reuse involves modifications of what is reused.
  - Template. Rather than reusing the actual content, one reuse the format of the deliverable in new deliverables.
  - As idea. In this case one reuse only some main ideas from existing artifacts, but do not reuse neither large part of the contents or the format of the original artifact.

It is usual to differentiate between methodologies being *for reuse* and those being *with reuse* [199, 396]. Another distinction is between reuse-in-the-large (e.g. reusing and fine-tuning whole applications) and reuse-in the-small (e.g. reusing selected objects, components, and functions).

8.1.5 Stakeholder participation

As also discussed in Chap. 3 stakeholders in CIS-support can be divided into the following groups [250]:

- Those who are responsible for its development, introduction and maintenance, for example, the project manager, system developers, communications experts, technical authors, training and user support staff, and their managers.
- Those with financial interest, responsible for the application systems sale or purchase.
- Those who have an interest in its use, for example end-users, indirect users and their managers.

We focus here specifically on end-user participation and define first some of the important terms in connection to this in more detail.

A **user** of a *CIS* is defined as a person who potentially increases his *knowledge* about some *phenomena* other than the *CIS* with the help of the *CIS*. An **end-user** increases his and hers *knowledge* in areas which are *relevant* to him independently of the actual *CIS* by *interacting* with the *CIS*. **Indirect users** increase their *knowledge* by getting results such as for instance reports or letters produced by the *CIS* without *interacting* directly with the *CIS*.

This is somewhat different from how 'user' is often defined, terming the system development and maintenance personnel as 'primary users' [169] or technical users. Not including these as users in the following discussion do not mean that they are not important stakeholders of the CIS support.

The term 'participation' means to take part in something. There exists different forms of participation:

- **Direct participation:**
  Every stakeholder has an opportunity to participate.

– **Indirect participation:**
  Every stakeholder participate more or less through representatives that are supposed to look after their interests. The representatives can either be:
  – Selected: The representatives are picked out by somebody, e.g. management.
  – Elected: The representatives are chosen by those being represented.

According to Heller [166], participation is sharing power and influence. He has divided the degree of influence and power into six categories as illustrated in Figure 8.1.

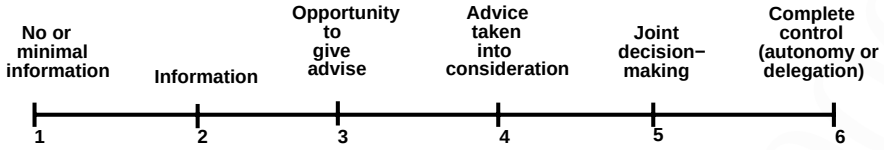


Fig. 8.1. Scale of influence and power

We would claim that participation when applied should be in categories 4, 5, and 6 on this scale, and will use this scale when classifying methodologies according to this aspect.

User-participation can be assured by the application of different links between the users and the system developers, some of which are used primarily under custom development where most of the functionality of the system is developed from scratch (C), and some primarily being used under packaged development, where most of the functionality of the system is delivered in a pre-packaged manner (P) [200].

Due to the large number of potential stakeholders in a development effort, in most cases representative participation will be the only practical possibility. From the point of view of social construction, it is doubtful that a user representative can truly represent anyone else than himself. On the other hand, even if the internal reality of each individual will always differ to some degree, the explicit knowledge concerning a constrained area might be more or less equal, especially within groups of social actors [136, 291]. Another factor is the scope of participation, i.e. when do participation take place. Usually one would expect that user-participation would take place heavily in analysis and in acceptance testing, more lightly in design, and very little in implementation, but this will often depend on the chosen methodology. When it comes to suggesting improvements of the current information system of the organization, direct participation should be possible. Also in the project establishment, a larger proportion of the stakeholders should be able to participate.

Many arguments for having participation have been given in the literature see e.g. [146, 268] for classifications. Here, user participation is basically

Table 8.1. Links between users and developers

Link	Used in P/C
Facilitated team (e.g. JAD [13])	C
MIS intermediary	C
Support line	P/C
Survey	P/C
UI-prototype	P/C
Functional prototyping	P/C
Interview	P/C
System testing	P/C
E-mail/bulletin board	P/C
Usability lab	P/C
Observational study	P/C
Marketing and sales	P
User group	P
Trade show	P
Focus group	P

motivated through a cost-benefit-perspective on the long run. Since all stakeholders have their individual local reality, everyone have a potential useful view of how the current situation can be improved. Including more people in the process will ideally increase the possibility of keeping up with the ever more rapidly changing environment of the organization. Added to this is the traditional argument of including those who is believed to have relevant knowledge in the area, and which are influenced by the solution. As indicated in several surveys, general participation appears to be a general indicator for (development) project success as perceived by all the different stakeholders. In Bergersen [27], the three most important factors for overall perceived project success were found to be the goal-setting, management support, and user-participation. In van Swede [377] the main contributions of success in the sense of satisfaction of all stakeholders were a cooperative environment, presence of a win-win starting point by considering the interest of all stakeholder-group, quality of project staff, and quality of project management.

Another aspect related to this point, is where updates of the CIS-portfolio take place:

- In the user organization.
- Centrally, with one unit developing the core of the systems, which are then customized locally.
- In a data department, developing customized systems.
- Externally developed packages with large local customization.
- By a different, known organization altogether (outsourcing).
- Externally developed packages with small local customization.

– By a set of partly unknown external organizations (e.g. by reusing components available on the Internet).

Typically, one would expect a mix of these models within the support of a portfolio. We will not investigate this in detail here. The possibility of active user participation typically diminishes as we go downward in this list of delivery methods.

8.1.6 Representation of product and process

Knowledge about the process and the product of CIS development and maintenance can be represented using both linguistic and non-linguistic means such as audio and video. Defined representational languages can be informal, semi-formal, or formal, having a logical and/or a executional semantics. Since we already have treated modeling aspects in detail in this book, we will not discuss this in anymore detail here.

8.1.7 Maturity

Whereas some methodologies have been used for many years by many organizations, others are only described in theory, and never tried out in practice. When discussing the maturity of a methodology, we can differentiate between the following factors:

– Is the methodology properly described? A description of a methodology is a model, and one can use the same approach as has been used in this book for assessing the quality of models and modeling languages in general.
– Is the methodology supported by tools(both for navigation, methodology adaptation and development and storing of artifacts)?
– Is the methodology used and updated through practical application? Is it used by many organization, supporting a large part of the portfolios in these organizations?
– Is the methodology undergoing a conscious evolution based on experience with it and scientific study of the use of the methodology?

Different parts of a methodology will often be of varying maturity.

Taking a philosophical standpoint neither reuse nor conceptual modeling nor having a defined methodology can be optimal, since all situations are unique, and thus in principle can best be attacked by using unique means. Reusing artifacts originally produced for some other purpose, in effect means to apply an externalization of the local reality of someone else than the current stakeholders, which thus can not be optimal. On the other hand, reuse is performed all the time. Using a commercial DBMS is for instance reuse, but it is not very wise to produce your own database management system when you perceive a need for this kind of functionality if you do not have

very special needs. Said bluntly, it is not very useful to use ten years to develop an application system, because one wants an optimal process of social construction. A balance between the different concerns brought up by our philosophical outlook is thus necessary

8.2 Conceptual Modeling in CIS Support in Organizations

As indicated in the previous section there is much more to CIS-support than conceptual modeling. On the other hand, conceptual modeling is looked upon as an important technique for reducing the gap between the user-communities local reality and computing technology supporting the social construction of the technology. Conceptual modeling is also looked upon as important for supporting both generative and compositional reuse both on a project and portfolio level over the life time of the system. This do not mean that we are not aware of other techniques for specifying requirements such as ethnographic techniques [141]. Where appropriate, also other aspects are mentioned including their links to conceptual modeling, but these will not be discussed in detail. This indicates that our suggested guidelines for a methodology is not in any sense complete. We are for instance not discussing project establishment which would be part of the planning activities and which could include many techniques from organizational development [74]. One specific technique found in organizational development literature that has influenced the description given below, is the search conference technique [107, 323]. This technique applies similar principles to participation that is used here. The technique has also been used earlier within application system development in Norway [370].We start with briefly discussing participation in some more detail.

8.2.1 Principles of Stakeholder Participation

Participation should ideally take place according to the process depicted in Fig. 8.2. The figure is adapted from a similar figure used in work in organizational development on participative action research [102] where a philosophical outlook that is similar to ours is used.

Before the outset of the project, a project establishment phase has been performed on some level and are taken as the starting point for the project. Users and other stakeholders all having their own internal reality and developers which also have individualized theories on the organizational reality and systems development and maintenance meets on arenas for dialogue.

The developers are more or less to be regarded as outsiders to the user community, either they are external consultants, or they are working in the CIS-department of the organization. Both the outsider's (i.e. systems developers) local reality and the insiders (i.e. primarily users) local reality are

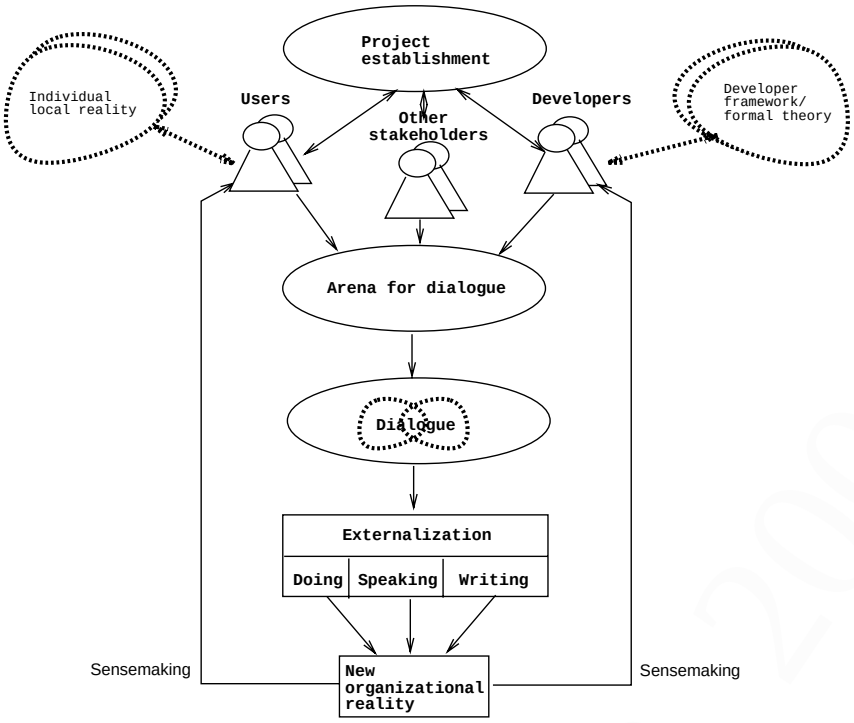


Fig. 8.2. Co-generative learning in systems development

regarded to be equally valid. This is necessary to avoid the problem of model monopoly in user participation where the system developers' or some group's perception of the world dominates the discussion, something that can result in what Habermas [154] terms 'naive consensus'. Bråten has developed the following guidelines [38] to avoid this:

1. Participants must be aware of the mechanisms of model monopoly.
2. One must have several different models available, and accept the validity of the individual models of each participant.
3. One must use time to develop an understanding of the participants own premises and models.
4. One must have an atmosphere that facilitates a democratic dialogue to eventually reach a common understanding of the problem, or at least an understanding of what the disagreement is all about.

Stated in our terminology, the last point means that one needs to achieve feasible comprehension and agreement on the constructed conceptual models.

In addition, it is important that all the participants are given training in the use of the conceptual modeling languages that are used. As stated by Heller [166], true participation necessitate competence. If the user par-

ticipants are supposed to be actively taking part in the model construction, they must have developed necessary skills in this in addition to having skills related to the application domain. This means that just being shown existing models that have been made by others for "validation" will not be sufficient, even when they are accompanied by intelligent explanations.

The participants will as a result of the democratic dialogues on given arenas externalize their reality in different ways. The externalization may result in both linguistic and technological artifacts, being first visible only within the project or for a limited group in the project. The linguistic artifacts will partly be in the form of conceptual models, which are one of the ways of externalizing ones local or shared reality through writing. Other linguistic artifacts can be made using natural language. Explaining these to others and arguing about them will be externalization in speech whereas when creating a prototype based on the conceptual model, others can use this for externalization of additional views through action. The models will be externalized and first be part of a larger organizational reality when committed to the organization in the form of an application system produced on the basis of the models.

From these externalizations the different stakeholders will be able to internalize the changes in their own local reality, and thus be able to discuss discrepancies and ultimately be able to apply the CIS effectively.

Overall Methodology. The approach to conceptual modeling that we will outline have some similarities with the approach developed for Tempora [367]. The modeling methodology is updated, including the use of techniques developed in PPP as described throughout the book to enhance the quality of the models and focusing on the social construction process.

Development of conceptual models is divided into several tasks, based on differences in the modeling domain:

- Development of conceptual models of the perceived current situation:
  The behavior, structure, rules, and actors in the problem domain including both computerized and manual data processing in the organization are described and analyzed. This is a description of the current organizational reality as it is internalized by the participants. A learning process about the current situation is meant to take place among all participants, in particular among future users and the developers of the system. The resulting model is often described as the As-Is model.
- Development of conceptual models of an perceived improved situation:
  This is also performed by first not taking a CIS solution into account, but is restricted by the environment that the project is performed within. In the end of this task, one specify specifically the external requirements to the CIS. The resulting model is often described as the To-Be model.
- Development of conceptual model of the future CIS:
  The behavior and structure of an application system are described and analyzed. Everything needed for an executable specification should be pro-

vided. This also includes the parallel and potentially intertwined development of a user-interface description. The models are late in this phase extended to include detailed design and computational semantics are specified. The resulting models ire often described as requirements specifications and design respectably
- Implementation:
  Based on the selected supportive technical actors such as the operating system and the DBMS, this is a more or less automatic translation of the design into an application system. The database of the application is populated, and the application is released to the organization.

Different project models can be overlaid the proposed approach. We will thus not present the methodology as a set of interconnected tasks although several such models should be possible to overlay the tasks described, making the approach more suitable for project management. Project management as such is regarded as being beyond the scope of the book. What we will present, is a set of general process heuristics linking the different modeling efforts up to the quality framework presented in Chap. 3. This overview is based on work performed by Krogstie and Sindre [345]. After this we will look in more detail upon a possible application of our conceptual framework in more detail concentrating on one of the modeling efforts: Developing conceptual models of the perceived current situation. In the end of this section we will outline the overall methodology for a set of projects.

8.2.2 Process Heuristics in Conceptual Modeling

Heuristics based on data which can be collected during modeling can be used to:

- Guide the actions of the current project.
- Guide the evolution of the organization's CIS-support methodology.

These issues can be called process support and meta-process support, respectively [181]. We will here focus on the first subject.

Overall Idea of the Process. In [239] the modeling process is vaguely described as consisting of cycles of expansion and consolidation of $\mathcal{M}$. In the following we will elaborate more on this idea, to prepare the ground for a discussion of process heuristics. Our idea of the process is illustrated as a state transition model in Fig. 8.3. The diagram can be explained as follows:

- **P - preparation:** In this state, the organization is performing actions in preparation of the modeling itself, e.g. selection of participants, training, and planning.
- **E - expansion:** Model statements are given, hence $\mathcal{M}$ is growing. During expansion, statements may be made more or less uncritically, i.e. thorough validation is not undertaken, and there might be errors introduced in $\mathcal{M}$.

Still, as long as some valid statements are made, $\mathcal{M}$'s degree of *completeness* will grow.
- **C - consolidation:** The model statements (especially those captured in the previous expansion phase) are consolidated with respect to *validity*, *comprehension*, and *agreement*, as defined in Fig. 3.
- **S – suspension:** The modeling activity is suspended. There may be several reasons for suspension. The model may have been agreed upon and "frozen" (e.g. to start the next modeling phase), or the project or a part of it may have been temporarily or permanently aborted.

This diagram consists of an inner cycle of expansion and consolidation, and an outer cycle including preparation and suspension. The starting state has been defined as **S**, i.e. before you do anything, you are in a state of suspension. The fact that there is no accepting state reflects the view that a computerized organizational information system is never finished.

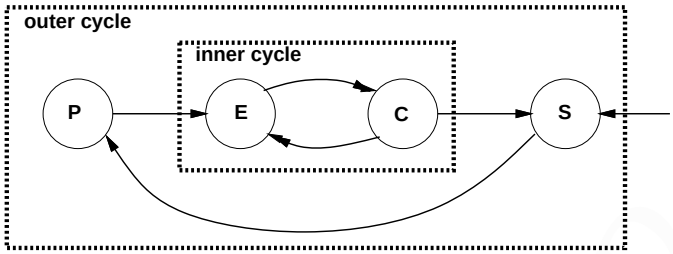


Fig. 8.3. The SPEC-cycle for modeling

Considering the entire modeling effort, there will be a lot of work in parallel by various participants. Hence, it is generally impossible to observe the simple state-changes of Fig. 8.3 for the model as a whole. It applies to smaller parts, the entire process then being composed of multiple such cycles.

The focus of the heuristics to be discussed here will mostly be on the inner cycle, in particular the switching between **E** and **C**. This switch can be based on the following:

- *Resource limit:* You are supposed to use a certain amount of time or manpower for **E**, then go to **C**, and vice versa, and similarly for **E** and **C** together vs. **S**.
- *Chunk size:* The number of statements made at one visit in **E**. When this size has been reached, there may be a policy to switch to **C**.
- *Progress:* You observe the progress made at **E** or **C** and switch when this has fallen below a certain threshold. The progress *will* decrease when the process has been in the same state for a while because of a phenomenon that may be called *exhaustion:* the most evident statements will be stated first, and the most evident errors found first. Moreover, staying too long

in **E** will yield a big chunk, for which incomprehension and disagreement is likely to hinder further growth of the model.

The three above strategies will be combined to make a practical process.

Gathering of Process Data. To start with, we will identify what data should be delivered to enable the heuristics. Then we will continue by briefly discussing how this can be obtained.

To determine the progress of modeling and the extent to which feasible quality has been reached, it is useful to know the point estimates for the quality goals, and their corresponding ratios (vs. resource consumption). In addition, model size has been included because it is important in the management of expansion, and model value because it is important in considerations about feasibility.

- perceived validity (PV), and ratio of perceived validity (PVR)
- perceived completeness (PC), and (PCR)
- perceived pragmatic quality (PP), and (PPR)
- perceived social quality (PS), and (PSR)
- perceived knowledge quality (PK), and (PKR)
- model size (MS), and (MSR)
- model value (MV), and (MVR)

The model size should be possible to obtain automatically. For the other data, one need to register:

- For all parts of the model, which have been perceived as complete (within their scope), which have been perceived as incomplete, and how big is this incompletion estimated to be.
- For all statements, which have been acknowledged as comprehended, which have been turned out to be incomprehensible, and which have not yet been checked.
- For all statements, which have been agreed upon and by whom, which have been disagreed upon and by whom, and which have not yet been checked.
- For each activity (here: each visit at **E** or **C**), how much resources are spent.
- For each activity, what is the perceived value increment to the model.

Some of these are easier to obtain than others. The most complicated are perceived completeness and model value. Even if these are dropped, it will be possible to provide some useful heuristics, as will be shown below. The phrase "for all statements" may give the impression that the above requires an enormous registration work. However, this need not be the case. E.g. discussing agreement of an ONER-model, it should be easy to implement functionality for quickly selecting larger parts of the diagram to mark it as agreed, open, or disagreed upon. Depending on the modeling situation (i.e. which languages that are used, and which parts of the languages that are used) one can get support from a CASE-tool to assess several of these based

on the knowledge of the modeling languages and metrics as indicated in Chap. 5.3.

Heuristics to Guide the Current Process. Heuristics will be presented as observed symptoms and possible actions. The list is not supposed to be exhaustive, and as one symptom may have several causes, there are several possible actions for each symptom. Heuristics must be evaluated by the participants in any specific case.

Expansion Heuristics.

Symptom E1: Resource consumption $\sim$ limit.
Symptom E2: MS increment $\sim$ recommended chunk size.
Action (E1 or E2) :

Switch to consolidation. For E2, if the recommended size was reached very easily, it might also be that the problem is simpler than assumed, so that it can be interesting to increase the recommended chunk size.

Symptom E3: MSR $<$ min.threshold (expansion is getting unproductive).
Symptom E4: MVR < 1 (growth of model value is less than resources being spent, i.e. work currently being done is perceived to yield deficit).
Actions (E3 or E4) :

- Switch to consolidation (if the problem is due to significant incomprehension or disagreement). If the chunk is well within the recommended size, it may also be sensible to lower the recommended chunk size.
- Switch to other techniques (if the problem is due to exhaustive use of some techniques and there are others which can be tried).
- Involve new participants (if the problem is due to exhaustive use of some participants and there are others which is perceived to possess relevant knowledge).

As shown here, an observed symptom from the collected information will not define a unique action; there has to be further considerations by the participants.

Consolidation Heuristics. Consolidation heuristics are more complex than expansion heuristics, since there are more goals and measures involved. We will avoid listing the most obvious heuristics. Hence, it is sensible to address pragmatic quality before validity, completeness, and agreement because comprehension of the model is necessary to achieve anything else with some certainty. Further, it is sensible to address validity and agreement before completeness. Guidance for this sequencing can be done by heuristics investigating the values PP, PV, PS, PC. This will not be discussed below. Instead we will look at symptoms indicating problems with the consolidation being done.

Symptom C1: Resource consumption $\sim$ limit.
Action: Switch to expansion (if resources available) or to suspension.

Symptom C2: PVR, PPR, PSR < min. threshold (i.e. consolidation is getting exhausted).
Symptom C3: MVR < 1 (i.e. perceived value being added to the model by consolidation is less than resources being spent).
Actions (C2 and C3) :

- Conclude that feasible quality has been reached (if the values for PV, PC, PP, PS are good).
- Terminate this part of the project as hopeless, or at least backtrack to some previous decision (if the values for PV, PC, PP, PS are bad and it is impossible to see any way out).
- Switch to expansion (if the value for PC is worse than PV, PP, PS).
- Switch to other techniques or to additional language training (if one or more of the values PV, PP, PS are unacceptable; concentrate on techniques applicable for the quality aspect which is most pressing).
- Involve other participants (if one or more of the values PV, PP, PS are unacceptable).

Heuristics to Guide Process Evolution. By aggregating collected information over several projects, one can also find heuristics to guide the evolution of the modeling-process. Important questions are:

- What specification languages, techniques, tools etc. seems to be appropriate for various categories of problems and for various stakeholders in the organization?
  Based on the user-modeling in connection with explanation generation, one can retain the information about which participants have comprehended what parts of which languages, for the use in later projects.
- What seems to be the optimal chunk size for various categories of problems using different parts of the modeling languages?
- What seems to be the optimal statement growth ratio for various kinds of problems using different parts of the modeling languages?
- What team constellations seem to be good for various kinds of problems?
- What kind of knowledge seems to be in shortage in the organization?

The results could be used to evaluate both approaches at the cycle-level and the composition of the entire process, i.e. suitable breakdown in cycles. In addition, post-project evaluations and evaluation based on actual usage of the system can be taken into concern. This is a large topic which is related to process improvement using frameworks such as CMM [300]. We will not go into details of this here.

8.2.3 Development Based on the Use of Conceptual Modeling

In this situation, one start with a specification base that is empty with regard to the application system that is of interest, and potentially a set of unsatisfied investigation reports. Due to the simplified situation that we look upon

here, the following will have many similarities to a traditional development project only taking one application system into account.

Preparation for Modeling Based on the project establishment performed during planning, the overall scope of the project has at this point been decided. On this background, one needs to identify $\mathcal{S}$, the set of stakeholders to the project.

In the conference system development project, the identified stakeholders were the organizing committee, the project group, the program committee, which would be direct users of some parts of the system and not only indirect as in traditional conference systems. Other potentially direct, but mostly indirect users were those receiving call for papers and call for participation, the subset of these being the contributors and/or conference attendants, the supporting conference organization at the university (SEVU), and finally the publisher of the proceedings (Chapman & Hall). For an extended project that would include making the system into a share-ware solution, several other potential stakeholders, i.e. potentially future program and organizing committees and different conference arrangement organizations and publishers could be identified. The overall actor-model for the conference organization is presented in Fig. 8.4.

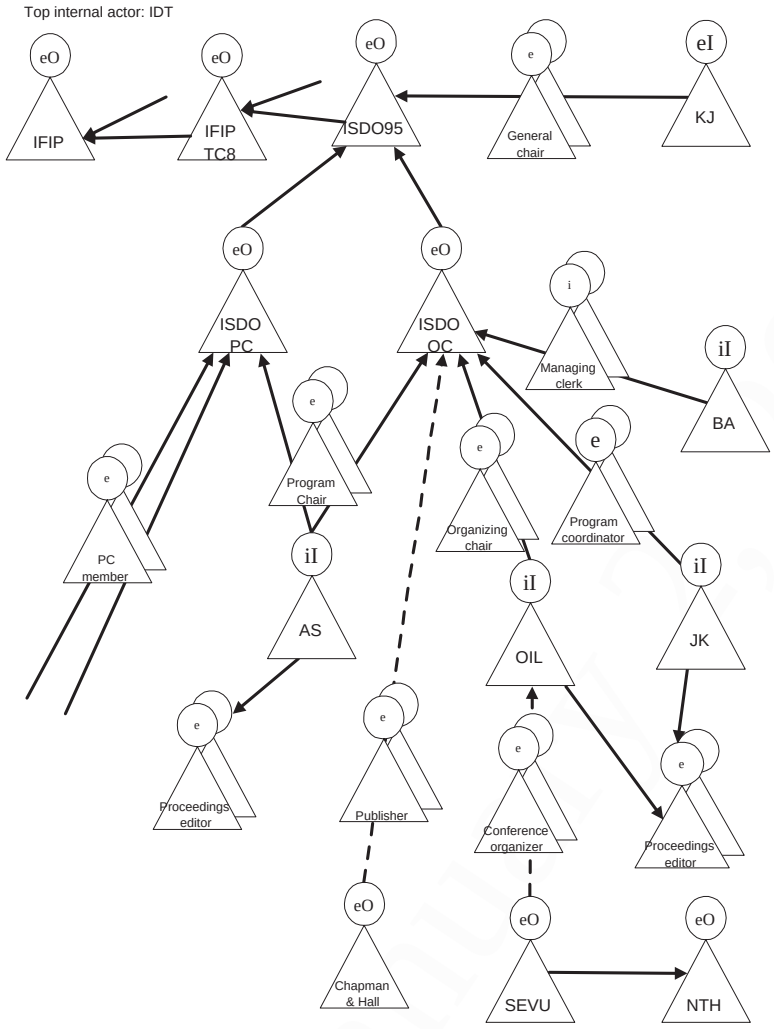


Fig. 8.4. Overall actor model of ISDO95

Having identified the stakeholders, it is important to select $\mathcal{P}$, the participants of the project: According to Mumford [267] user-participation works best with a two-tier structure of a steering committee and a development group. This kind of organization is also often used when applying more traditional methodologies such as Method/1 [9]. The steering committee will set the overall guidelines for the development group when it comes to the use of resources. It's members will typically include senior managers from affected user areas, senior management from the CIS-department and senior trade union officials, at least in Scandinavia. The development group will consist of representatives of all major groups of stakeholders of the project. All in the design group except system developers should be selected or elected in a way that is acceptable to their constituents and seen as democratic. One should be especially careful to avoid the situation where a group of end-users are 'represented' by a middle-level manager which himself has no intimate knowledge of the detailed tasks to be supported by the CIS [200]. Seen from the point of view of social construction, we recognize that the internal reality of each individual is necessarily internally held and hence reflect individual variation. It is nonetheless useful to distinguish those elements that through socialization, interaction, or negotiation individuals have in common. It is these collective cognitive elements that individuals draw on to construct and reconstruct organizational reality. Gjersvik [136] terms this a local reality on a group level. This is not the sum of the individuals local realities, nor is it shared group reality. The group's local reality is a way of acting in relationship to the organization. The way of acting is developed inter-subjectively among the individual actors of the group. Examples of groups holding a local reality can be the members of a department, the managers, or the union members. They develop this group local reality because they have common work experiences, and because they identify with each other in the organizational context. In his case-study Gjersvik identified four distinct group realities: Managers, supervisors, shop floor workers, and the union. Relevant dimensions in such discussions are:

- Organizational level.
- Department.
- Educational background.
- If they traditionally are decision makers or not.
- Experience in the organization.

Mumford claim that there should be a representative from each major section and function, each grade, age group, and sex if possible, in addition to the system developers.

In addition can geographical location, experience with computers and similar applications and methodology, and attitudes to change be used as dividing lines. Using such characteristics, techniques such as QFD [258] identifies roles, which are aligned with the project success factors. In this way, one are also able to assess the importance of the different roles, and thus the

importance of the views and requirements of actors representing these roles for overall project success.

Examples of users types that are specifically important include:

- They make up the majority of users for the application.
- They shape company opinions or attitudes.
- They were particularly disappointed with the prior system.
- hey have financial responsibility for the project.

In the conference support system development project, the "steering committee" consisted of AS[1] and a student supervisor MH. The system developers will be represented as group GR, which consisted of students in the 4.year at NTH, having mostly a similar educational background. Although individual differences obviously existed, we will for simplicity treat them as a unity. In addition, JK and OIL functioned as users to the system, JK being the primary end-user, at least of the first version of the system. AS also functioned as a provider of information and as an indirect users. JK and AS was identified as the most important users to involve extensively. The PPP CASE-tool is the main technical actor of the audience. The main social actors in the project and their relationships are given in Fig. 8.5. All actors being part of the project group in some role is denoted as internal actors in this model.

$\mathcal{A} = \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}, A_{11}, A_{12}, A_{13}\}$

where A_1 = AS, A_2 = JK, A_3 = OIL, A_4 = JPH, A_5 = YJ, A_6 = KPH, A_7 = HFS, A_8 = SG, A_9 = EA, A_{10} = FVL, A_{11} = GR, A_{12} = ISDO, A_{13} = PPP.

GR = {JPH,YJ,KPH,HFS,SG,EA,FVL}
ISDO = $\mathcal{P}$ = {JK,OIL,AS,JPH,YJ,KPH,HFS,SG,EA,FVL}

Three distinct realities could be distinguished. The system developer group GR, a mature professor AS, and two young researchers connected to the research group of AS, JK and OIL which had a comparable background, although with different requirements to a system of this sort.

To simplify, we can say that at the start of the project the main involved actors had the following relevant knowledge:

- General system development knowledge $\subset \mathcal{K}_{11}$
- { General system development knowledge, PPP knowledge, Overall conference organizing knowledge } $\subset \mathcal{K}_1$
- { General system development knowledge, Specific knowledge on WWW and Ingres, PPP knowledge, Specific conference organizing knowledge } $\subset \mathcal{K}_2$
- { General system development knowledge, PPP knowledge, Specific conference organizing knowledge } $\subset \mathcal{K}_3$

[1] For brevity and privacy, we only indicate the initials of individuals.

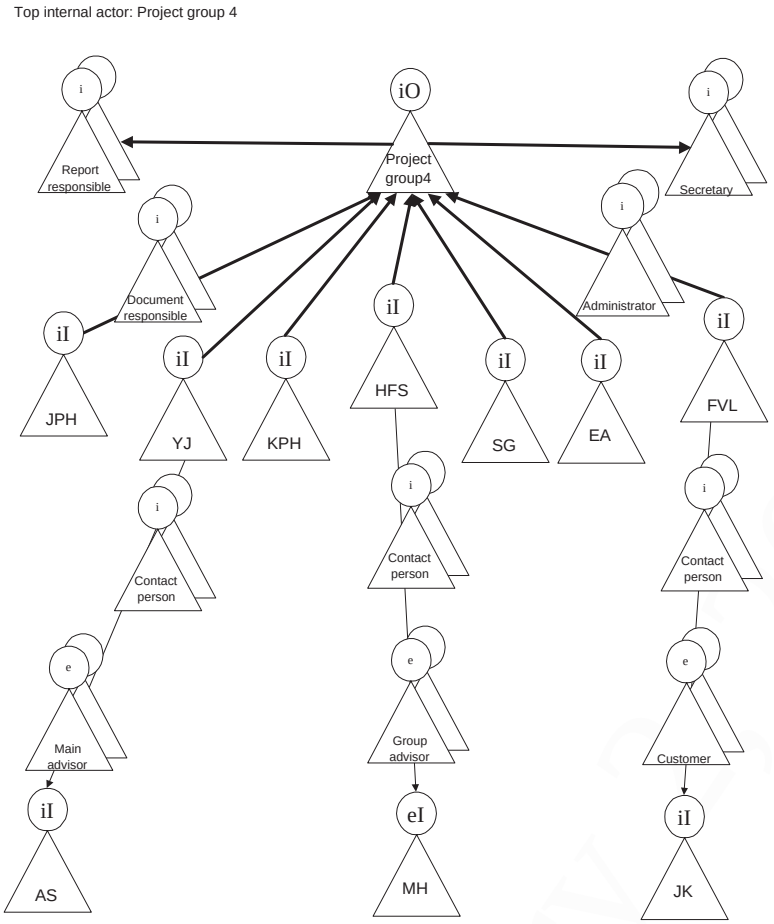


Fig. 8.5. Actor model of project-participants

Before starting on the modeling effort, a preparation period to organize and plan the project take place. This period also usually includes an initial learning course for the participants, depending on their current knowledge of the modeling languages and the domain. Whereas the system developers might use traditional ethnographic techniques such as on-site observations of ongoing activities, open-ended interviews, mappings of who is doing what and where, exploration of roles and responsibilities, identification of tasks, goals, and chains of accountability, and examinations of manuals, job descriptions, directories, division charts, and other organizationally related documents, the user-representatives will normally need training in the methodology and its purpose, in addition to the use of the conceptual modeling languages. The training at this point should concentrate on the most important parts of the languages, so that the participants are able to model the most common sit-

uations. Learning the more advanced parts should be done on an as needed basis during the project [132]. If a user-model is to be applied in connection with explanation-generation this can be initialized at this point.

In the conference case, the project group looked upon existing material describing professional conferences, and took contact with SEVU to get more information regarding the practical arrangements of a conference, since it appeared that this kind of knowledge was lacking among the original participants. Thus, one had here an example of inter-subjectively agreed knowledge incompleteness that was remedied by contacting external experts in the field.

Another part of the preparation phase is the preparation of arenas for dialogue and modeling. This can be both the preparation of technical solutions, i.e. live-boards, meeting-room system, or other co-located or distributed, synchronous or asynchronous groupware solutions according to what is deemed necessary and is available. Another possibility is to use rooms with plastic walls which one are able to write/draw on or can paste and remove paper on in order to illustrate different aspects as has been used in e.g. Tempora and in the ABC-method [397] for preliminary modeling. These models can then be transferred to a more persistent medium such as a CASE-tool to increase the physical quality of the model on persistence and distributability and to enable further advanced treatment of it.

At this point, selection of which languages to use at which stages are also partly decided. In our case the total set of languages to use for conceptual modeling is more or less given, i.e. $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3 \cup \mathcal{L}_4 \cup \mathcal{L}_5 \cup \mathcal{L}_6$ where L_1 = ONER, L_2 = PPM, L_3 = DRL, L_4 = Rule relationships, L_5 = the Actor modeling language, and L_6 is the set of links between model elements in the different languages.

Below, more general guidelines for which languages and part of languages that have been found to be applicable at each stage are presented. The guidelines are partly based on practical experiences from the Tempora-project.

In the conference support system development project, the following was decided upon:

- Development of conceptual models of the perceived current situation: Here modeling of a typical conference of this sort was performed. Languages to be used: Informal rules, rule-hierarchies, the actor-modeling language, and PPM including ports. Limited use of speech acts modeling.
- Development of conceptual models of the perceived improved situation: Modeling the same aspects as above, but with the understanding that some of the functionality should be computer supported. Applying the same modeling languages as above except speech act modeling, but develop the models in more detail. Also include ONER-modeling. Only updating the actor models if necessary.
- Development of conceptual model of the future CIS: Decide which processes should be supported by the CIS, and model these in more detail including

the modeling of DRL-rules and PLD's as appropriate, in addition to further formalizing of the other models.

As will be seen below, this is somewhat different from the guidelines below, which indicate that one must apply modeling languages and techniques according to the specific situation. Specific to our situation was the thorough knowledge of the modeling languages by the users

Modeling of the Existing Information System (EIS). As a starting point for modeling, the root-definitions [61] for all participants are established using the CATWOE-technique. The use of this for the conference support system development project was described in the previous chapter, and is not repeated here.

Starting out with the CATWOE-analysis is deemed important to enable the construction of individual models for the participants. When new persons are included in the audience, at a later stage also their root definitions should be established. The CATWOE-analysis gives starting points for several models. It might define actors and roles more clearly. Actors and roles are also found during the stakeholder identification, but additional actors and roles might be identified here. We will discuss further modeling of actors and roles primarily as part of the process and data modeling described below because of the close links between these. It can be a start of process models, indicating the main activities, and one can also use them for identifying major entities for an ONER-model. Most importantly, some of the high-level goals of the different participants which can be used initially in the goal-hierarchy are established through the CATWOE-analysis.

For the further modeling, the development group is first divided into homogeneous groups for the first modeling efforts similarly to how the division is done in e.g. search conferences. This is contrary to what is done in e.g. ABC and JAD [13] where the use of inhomogeneous groups is proposed from the start. The main argument for homogeneous groups which in one extreme include one user-participant and in the other extreme all the participants, is to allow time for the individual to develop his own model to address problems of model monopoly. This apparent redundancy is also important for knowledge creation [275]. In the case-study of Gjersvik [136], the local reality of management was more easily transferred into an application system because of their training in thinking in abstractions due to their generally higher education. Problems of this kind with JAD have also been reported in [55].

In any case each conceptual model "fuses" at least two internal realities. If the users had been excluded from the analysis process, the system developer would have had to bridge both his own and the reality of someone knowledgeable in the domain. The situation is symmetrical for users. If they undertake the development themselves, they must cope with the presumptions that is embedded in the modeling languages and tools. If users develop also their own languages in every case, reuse will be minimal, and it will be difficult

to support model and application system quality. We suggest to have a user and a developer to be the minimum unit of cooperation, where the developer will be supposed to be able to use a larger part of the conceptual modeling languages to be able to apply techniques to improve semantic and pragmatic quality of the models. We will below assume that more than one model is created based on the local reality of parts of the participants, thus that there will be a need to merge different models, a process which is supposed to include both negotiation and mutual learning.

In the conference support system development project, models were first developed individually for actors, thus developing $\mathcal{M}(EIS)_1$ as an externalization of $\mathcal{K}_1$, $\mathcal{M}(EIS)_2$ as an externalization of $\mathcal{K}_2$, and $\mathcal{M}(EIS)_3$ as an externalization of $\mathcal{K}_3$

The model-integration strategy was originally to first integrate $\mathcal{M}(EIS)_1$ and $\mathcal{M}(EIS)_2$ into $\mathcal{M}(EIS)_{1,2}$, and then integrate $\mathcal{M}(EIS)_3$ with $\mathcal{M}(EIS)_{1,2}$ into $\mathcal{M}(EIS)_{12}$.

To assure organizational learning $\mathcal{M}(EIS)_{12} \subseteq \mathcal{M}(EIS)^1, \mathcal{M}(EIS)_{12} \subseteq \mathcal{M}(EIS)^2$, i.e. whereas JK and AS had interest in the complete model, OIL had only interest in the part that was an externalization of his knowledge regarding the practical organization of the conference.

Although education of the participants has been performed on the use of parts of the languages that has been selected for use at this stage, different persons should use different languages according to their ability to deal with abstractions. Experience indicate that it might often be easier for workers to tell their stories using the parts of the process model which are able to deal with the instance level. These instance-level stories can then be abstracted up to class-level ones. A data-model on the other hand represents only class-abstractions. For others, e.g. management, which traditionally will be better suited to deal with abstractions, a data modeling language can be more suitable for constructing models [136]. By using driving questions though, and by applying the links between models actively, it should be possible for all participants to build up models in the different languages that are used and integrate them.

As an example of the modeling on the instance level, Fig. 8.6 indicate the situation regarding the distribution of CFP's seen from the point of view of the actor JK.

This could have been based on the following story.

> "The other day, I got a request from NN to send him the CFP for the conference. The CFP had I earlier got from AS who originally made it. After updating the address-list over people that wanted to receive information about the conference, the CFP was sent to NN"

This model can then be generalized and extended as depicted in Fig. 8.7. The main generalization is achieved by transferring actors into roles.

The modeling on this stage is divided into five main tasks. That does not mean that the tasks are done sequentially, one typically will go back and forth

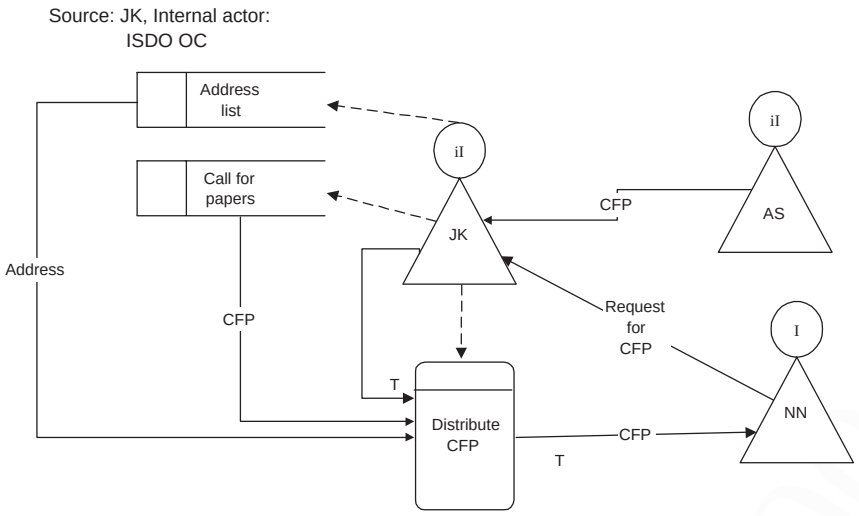


Fig. 8.6. PPM describing CFP-distribution at the actor-level

between different tasks. We will first describe the general guidelines, based on both the case studies and on guidelines developed for Tempora, and then indicate briefly how modeling was done in the conference case. The tasks are:

- Goal modeling.
- Data modeling.
- Process modeling including actor/role modeling.
- Rule modeling.
- Using techniques to help increasing the pragmatic, perceived semantic, and social quality of the models.

Goal modeling. Objectives as perceived by the participant are described in connection with higher level objectives e.g. overall strategies. The start of the goal-model is based on the results from the CATWOE-analysis as illustrated above. As goals and rules are identified (see rule-modeling below), one try to incorporate them in the goal-model using the deontic relationships.

Both for the individual models and the integrated model, rule and goal-modeling was used in the development of the conference system.

PPM modeling. PPM modeling by non-technical participant should be performed without applying ports and not from the start distinguishing between triggering and non-triggering flows. Thus the PPM as used here resembles traditional DFD. The modeling of process logic should not be considered. On the other hand, if it is necessary for the comprehension of the model, the developer might on parts of these models use all of PPM including PLD or rules to enable execution, code-generation, and explanation. When high pragmatic quality has been achieved, one will continue modeling temporarily

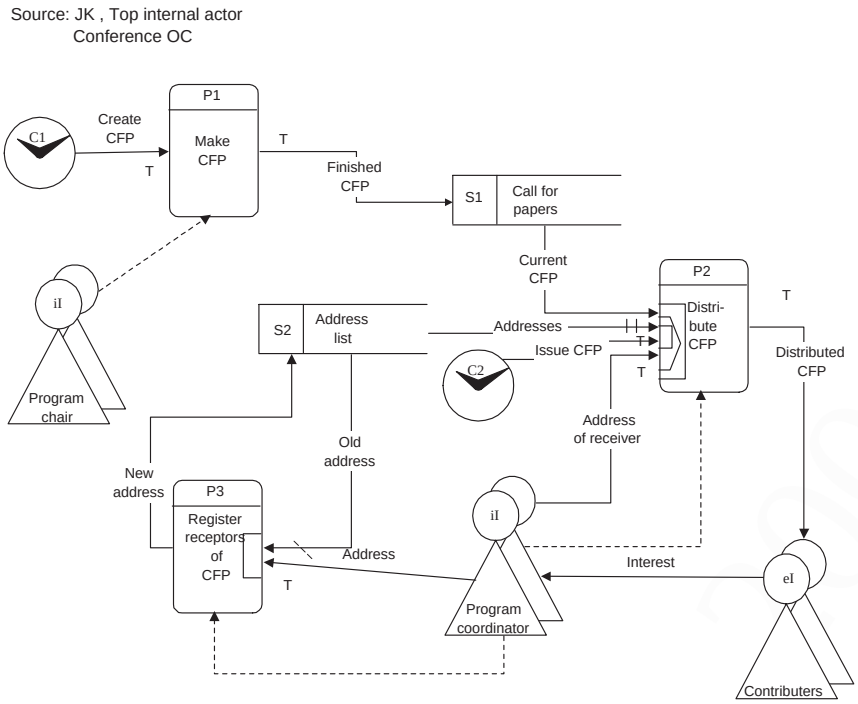


Fig. 8.7. PPM describing CFP-distribution at the role-level

discarding the extra details added either by using an appropriate filter or by backtracking to a previous version of the model.

Guidelines:

- The process modeling do not necessarily start from a top-level process, i.e. it is more convenient for certain groups to deal with the detailed processing in one area. This in contrary to the idea of developing a top-level context-diagram as described [129].
- A process corresponds to a business process, not an automated process of the target CIS. An actor or role can be used to indicate in more detail who supports the process.
- The scenario to the ONER-model if any such model is developed at this stage conveys an abstraction facility for all kinds of stored data. One know that the data is stored in some form but a further description is post-poned. The contents of ONER-scenarios may be specified by simple access expressions or even left out.
- Flows at a high level of decomposition are not detailed until the process model at lower levels of decomposition are well understood.
- In the beginning, no distinction is made between control flow and data flow. Then one concentrates on the identification of control flows.

- Filtering is useful to support the construction of port structures. This is particularly true when the number of flows entering or leaving a port is high.
- Each process must at least be associated with one event-action rule, although this can be described informally or in matrices.
- Flow content can be specified with regard to an ONER-model, but this is not mandatory.
- Actors and roles in the diagrams can be either external or internal to the organization. One do not apply the traditional notion of 'external entity' at this point, where external refers to being outside the system boundaries.
- If one are modeling speech-acts, one should try to complete the conversations, and structure the process models around the conversations.

In the conference support system devlopment project, one initially developed process-models in the DFD-style, with the addition of indicating supporting actors. Since the participants knew PPM-modeling well, also triggering and terminating flows and ports were indicated. Models where made on smaller areas, trying to synthesize an overall process-model through model-integration. No ONER-model was made at this point, thus stores and flows were not described in much detail. Limited speech act-modeling was done separately.

ONER modeling. Concentrating on the modeling of main entity-classes and relationship-classes, including attributes and values if necessary. Cardinality constraints are specified if it is deemed necessary. Similarly to above, the system developer might include additional model-details to be able to use the techniques for improving the model-quality.

Some guidelines at this stage are:

- Strict classification of entities into specialization hierarchies should not be enforced early in the modeling process. Therefore, one should postpone modeling of specialization of entity classes if they do not have any particular relationship classes to any of the other central components of the model, or they appear naturally in the discussion of roles in the organization. Alternatively, they are hidden or developed in separate views.
- Different views of the same phenomena are allowed, since different properties may be of interest in different situations.
- Redundancy and division into submodels are often necessary to increase pragmatic quality.

DRL modeling. Most rules are to be expressed in natural language, using a general *when-if-then-deontic-for-role/actor-consequence-else-consequence* structure using informal text in the fields.

Guidelines:

- Both informal and formal rules should be explicitly represented.
- If the number of rules specifying exceptions to a rule is high, the specialization criterion of the entity classes involved should be reviewed.

- Predicates are useful to split a composite rule into simpler rules.
- Ports together with abstraction mechanisms are useful to support the rule formulation when the number of parts making up a rule is high.
- The I/O-matrices can be useful to support rule formulation when there is a high number of combinations to be taken into account.

When developing the conference system, rules were only informally described at this point.

Increase model quality. In the expansion-phase of modeling, the driving-questions technique can be performed, often not even using a modeling tool in the first expansion/consolidation-cycles. When improving the physical quality of the model by transferring it to a modeling tool, only syntactic invalidity is addressed immediately by the tool.

Before switching from expansion to consolidation, checks for syntactic incompleteness can be performed, before concentrating on comprehension. In the conference case, comprehension techniques at this stage were inspection and filtering. This was judged as sufficient because of the comprehensive knowledge of the modeling languages held by the participants. This would not normally be the case and one could here be assisted by the explanation-generation facilities to become familiar with the modeling languages. One could also at this stage have added more details to the model, enabling execution and explanations of the model behavior to be generated. Based on the knowledge of what needs to be added to different kinds of models to use the different techniques, one could get an indication of the "incompleteness" of the model with regard to execution or explanation generation based on the area selected to be executed e.g. what is missing for the model to have necessary formality as discussed in Chap. 3. Based on earlier experience one could then get an indication of the resources needed for preparing for a prototype or explanations.

After comprehending the different models, perceived validity and completeness can be looked into. The driving questions can be used also here, but in a way to get an indication of that there are incompleteness, rather than to start on a new expansion-phase immediately. One should also apply techniques for consistency-checking of the models here to support this process. If the perceived semantic quality is regarded as satisfactory on this level, one can return to a reconciliation of the models written in the different languages to check such things as consistent naming before trying to integrate models based on the views of different individuals. Throughout modeling one should also build up a thesaurus of terms which are negotiated in parallel to the development of models [31]. As an example of a process model made in this way, Fig. 8.8 shows one of the PPM models made based on $\mathcal{K}_2$.

When integrating models made by different actors, this is one area where model integration techniques can be useful. Totally automatic methods are not applicable in this case, merging will necessarily involve human judgment and negotiation. When comparing models which have no common predecessor

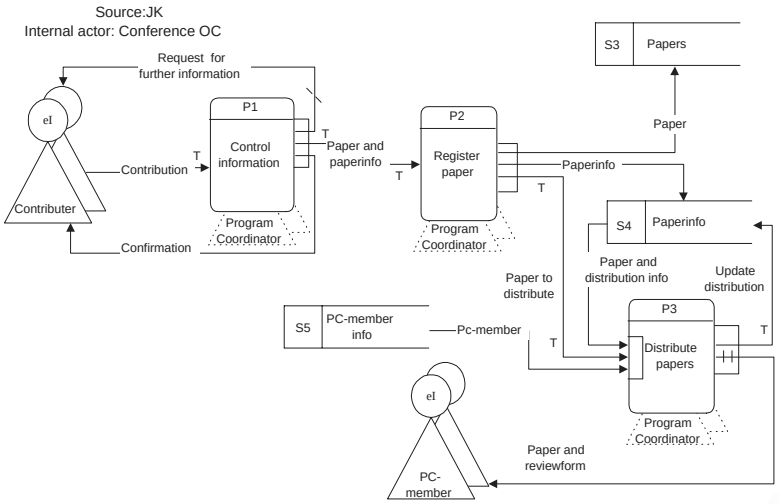


Fig. 8.8. PPM describing reception and distribution of papers based on $\mathcal{K}_2$

made by persons from different parts of the organization, it is usually more of an issue to discover overlap than differences between models. Before doing negotiations to get a common model, the comprehension of the models made based on the internal reality of other participants must be assisted. If all models are made in languages that all involved participants are familiar with, one can start by having the participants trying to explain the models made by others. In this case it might be more cost-efficient than above to add details to parts of the models to support comprehension techniques such as execution and explanation generation.

If the models are developed using parts of the languages that the other participants have not yet used themselves, language training in this area is given in the preparation phase. Initially the other models are regarded as fully uncomprehended, but totally valid, i.e. $\forall i, j \; i \neq j, \mathcal{M}_j \setminus \mathcal{I}_i = \mathcal{M}_j$. Comprehension is then incrementally achieved and validity is questioned in the process. In this way the participants can learn about other parts of the organization and how others perceive it at the same time as they attempt to externalize their local reality. When comprehension is achieved, model integration of different models are attempted. This process might result in the discovery of conflicts. Of special interest is the discovery of inconsistencies, which might need to be reconciled.

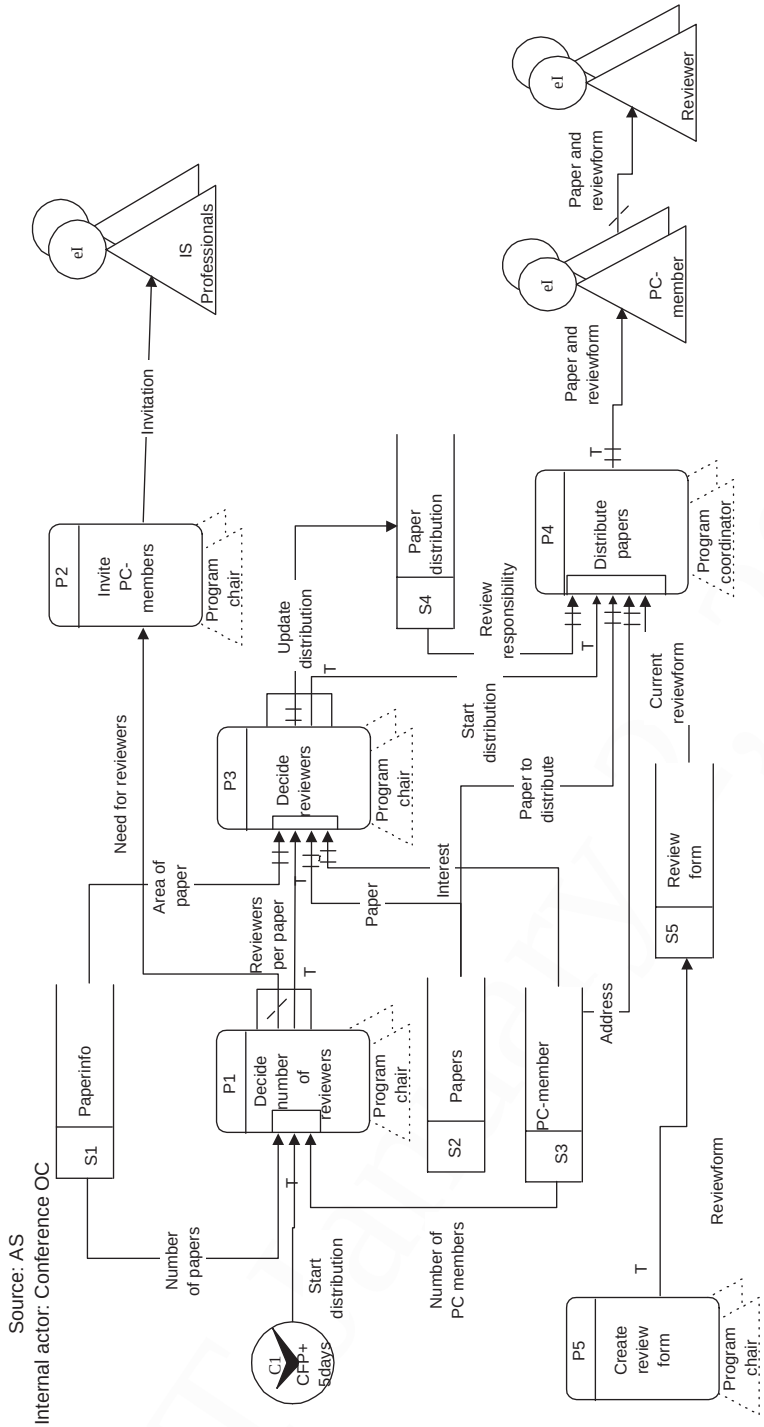


Fig. 8.9. PPM describing distribution of papers based on $\mathcal{K}_1$

An example from the conference support system development project is the comparison of Fig. 8.9 where the source was AS with Fig. 8.8. We can recognize one common process (Distribute papers) and one common role which receive papers (PC-member), whereas the additional processes in the diagrams regard different areas. Also some common data-stores can be recognized using similar, but different names. How this process interact with other processes differs between the models, and needs to be reconciliated. The issue is if one should distribute papers to reviewers on a paper to paper basis as they are received from contributors as suggested by the model of JK, or as one process which is first run after that all papers have been received, as suggested by the model of AS.

After integrating the models, one should look upon the amount of the resulting statements in the combined model that was the source of each of the participants, comparing this with the number of statements in each of the original models. If only a small percentage of the statements in the original model of one or more of the participants are retained in the official part of the joint model, this might indicate a situation of model-monopoly. The resulting model might also be perfectly legitimate of course.

After model-integration, one might go back to a phase of expansion based on the new combined actor, which are followed by a new consolidation phase. Alternatively, one might like to reconcile this model with another model right away according to the chosen integration strategy.

These kinds of processes will continue until there is a common model of the participants perception of the current situation. In a sense this has created a new organizational reality for the development group, although it do not at this point apply generally in the organization. This do not mean that all differences in the models are removed. It is neither to be expected or wanted that even after a mutual learning process, the participants look upon the organization in an identical way. The overall assumptions that surfaced in the CATWOE-analysis have not necessarily changed. The representation of this variety is most easily done in the goal-model with its possibilities of explicit representation of inconsistencies between views. Since the goal-model also links to and motivates statements in other models, it will be the main arena for negotiations.

In the conference support system development project, models based on the knowledge of JK, AS, and OIL were created. Then, the models made based on the knowledge of JK and AS were reconciled. At this point it was apparent that one needed to concentrate on the paper-process only, and thus a further reconciliation of this combined model with the model based on $\mathcal{K}_3$ was not performed, although it was planned. A reconciled overall process model of the paper-process is given in Fig. 8.10.

In [207] we have also described the other modeling tasks in a similar manner as above, but this is not included in the book for brevity.

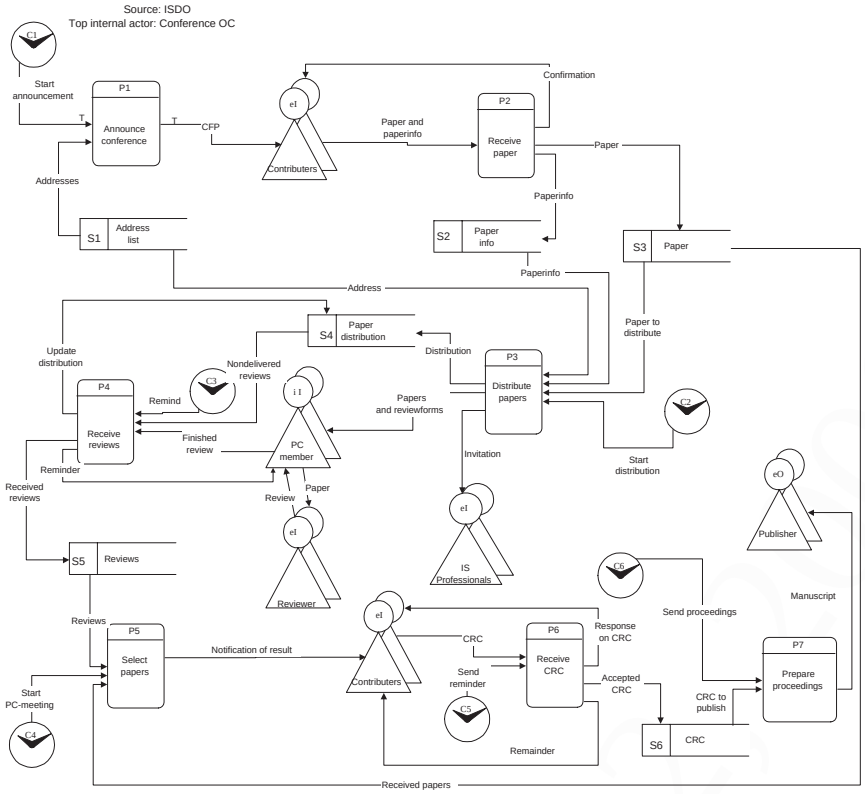


Fig. 8.10. PPM describing the paper handling

Summary on Modeling Approach. Figure 8.11 can be used to summarize the approach, giving a generic overview of the conceptual modeling within a project. Each circle indicate a modeling effort based on the knowledge of an individual or organizational actor. As indicated in the top left of the figure each modeling effort can go through the modeling of several revisions and variants of the model, and the effort follows the SPEC-cycle, ending this modeling effort at suspension. We will look in more detail on mechanisms for versioning support and development transactions in the end of the chapter. Any modeling cycle in the same or previous major phase may be restarted e.g. in case one use an iterative development strategy. It is also indicated that a modeling effort can use a limited set of the overall conceptual framework, and use this in different depth [176].

When integrating two (or several), models, their own modeling efforts are in the suspension state, and the modeling of the joint model moves to preparation, setting up for merging and further modeling for a joint organizational actor, and possibly additional language training. When whole or more usually parts of the existing models are introduced, this is the first expansion

activity, which immediately is followed by a consolidation phase, where one try to achieve pragmatic, perceived semantic, and social quality. One can then return to further expansion/consolidation cycles on this model. In this model, one will typically retain statements from all the predecessor-models, and also prune statements from all the predecessor models (indicated with black in the figure). In addition will often new statements be added through this modeling effort, indicated in the circles by the area below the black area.

When the modeling of the perceived existing information system is suspended, one will take whole or a subset of this model and bring it over to FIS-models. This subset can possibly be further partitioned. A similar process takes place on the FIS and the FCIS level as on the EIS level, before a CIS is created based on the final FCIS-model, and manual procedures based on the FIS-model is externalized, and this is all committed to (part of) the organization, resulting in a buffered transition of the COIS. One can also perceive a situation where several CISs for different parts of the organizations are created and committed to the different parts of the organization. In any case, the actor models should be updated to show which parts of the new CIS that are accessible by which users, using detailed support relationships. The overall pre-integration strategy should be made in early project planning, but should be updated as found appropriate during the project.

In addition to the creation of a CIS, the knowledge of the participants should have changed as part of the project. In the conference support system development project the developer group learned about the arrangement of professional conferences in addition to conceptual modeling and specific technology such as Ingres DBMS and WWW. Both AS and especially JK learnt about these same technologies. In addition, and equally important, JK learnt a lot about the arrangement of professional conferences, thus one do not only have the potential for mutual learning [221] where developers learn about the application area, and users learn about new technology, but also for organizational learning, where different users learn from each other as part of participating in the project.

In a maintenance project, the newly externalized CIS and manual procedures are taken as one outset, that are used together with the earlier developed EIS-model, and based on this a new model of the existing reality as it is perceived can be created. A similar process of sub-setting, splitting and merging as for the development project can be conceived, but we have not indicated this in the figure. We have included existing COISIRs as being taken into account on FIS and FCIS-modeling though. When the new version of the CIS is finally committed, this is a new buffered transition of the COIS.

8.3 Management of Change

We end the book with an overview of some mechanisms for configuration management of conceptual models

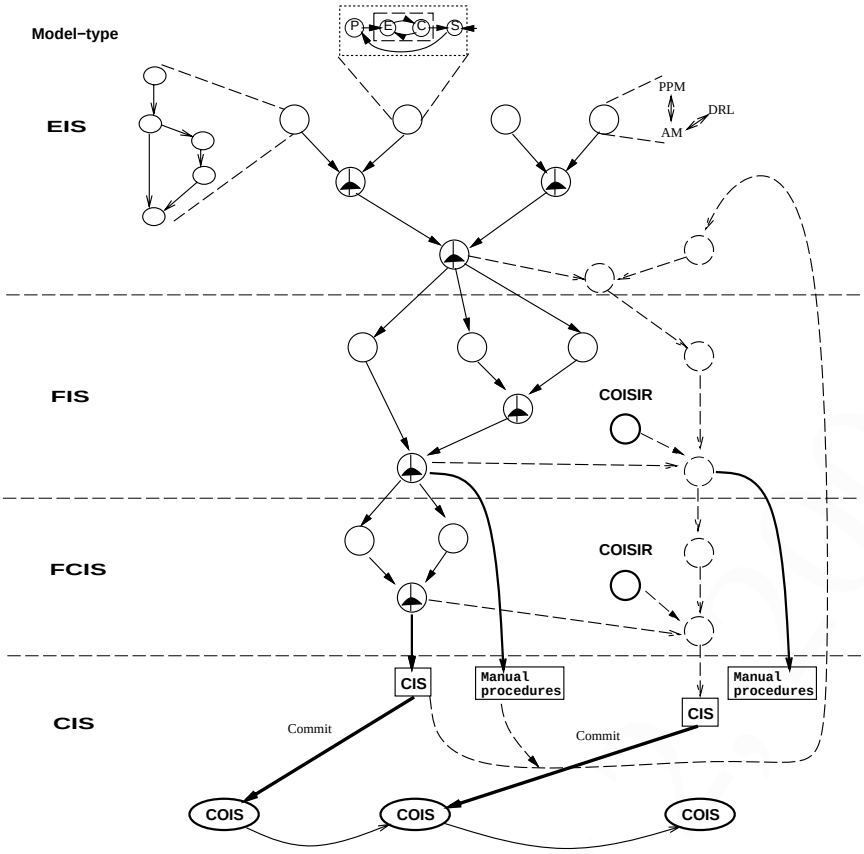


Fig. 8.11. Conceptual modeling in a set of integrated projects

This section presents necessary functionality to support the change process during information system development. This includes a framework for version and configuration management which makes it possible to model the various product structures involved in information system modeling, and which permits inconsistencies in the information system models. It also includes augmentation of the framework to support development transactions which allows asynchronous modes of working and coordination within the development team. Finally, it includes further augmentation of the framework for supporting synchronization of the work performed by the individual members of the development team.

8.3.1 Version and Configuration Management

Version and configuration management keeps track of versions of system components developed by several developers potentially working in a geographi-

cally distributed development environment. A version of a system is an immutable, identifiable edition of a system. A system is composed of a number of components. A component may either be a hierarchical composition of other (sub-)components or a flat structure with no hierarchical relationships among the (sub-)components. A version of a system is a composition of versions of system components. The following issues are discussed: Definition of development object versions, definition of component structure, naming of components, and granularity of versioned components.

Definition of Development Object Versions. Each development object may exist in a number of versions. There are two classes of versions: Revisions are versions that are meant to replace previous versions, and variants are versions that are meant to coexist with previous versions. The difference between two succeeding versions can be characterized by a set of elementary (atomic) changes. We take the assumption that each version is developed based on a unique existing version. A version of a development object is named as follows: development-object-name revision-number[variant-name]. Revisions are given a two digit revision identifier in addition to their name. The two digits make it possible to distinguish between major and minor version changes. Variants are given a variant specific name in addition to development object name and revision identifier.

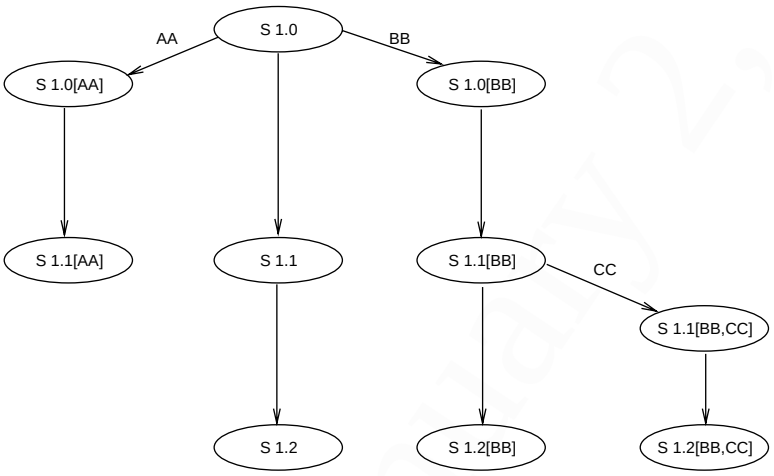


Fig. 8.12. Version graph for system S

Version graphs can be used to graphically depict the relationships between versions. Figure 8.12, depicts a version graph for a generic system S, which exist in ten different versions. The initial version is S 1.0. Two variants, called S 1.0[AA] and S 1.0[BB] respectively, have been developed based on S 1.0. Further, version S 1.1 has been developed as a revision of S 1.0. Detailed information on creation time is also associated with each version.

Definition of Component Structure. A system comprises a number of system components. The components are either a hierarchical composition of other (sub-)components or a flat structure with no inherent hierarchy. An example of a hierarchical component structure is a set of Data Flow Diagrams which are hierarchically ordered by way of decomposition relationships from processes in one model to other (sub-)models. An example of a flat component structure is an ER model where there are no hierarchical relationships among the different components of the diagram. The two types of component structures require different treatment with respect to versioning.

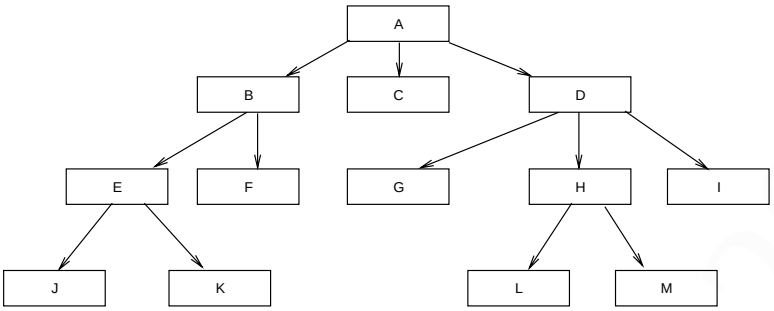


Fig. 8.13. General hierarchy

Hierarchical component structures. The hierarchical models are characterized by hierarchical decomposition. Decomposition of a component is a refinement of the component in terms of a more detailed structure of lower level, i.e. more detailed, components. Hence, every system component is either atomic or compound. An atomic component has not been further decomposed, while a compound component has been further decomposed into a set of system components. Every system component exists in a number of versions. Thus, a specific version of a non atomic component is composed of specific versions of its subordinate components. In a standard hierarchy a modification in a leaf node component, which will create a new version of that component, would result in a ripple of version changes all the way to the top of the hierarchy. Consider the hierarchy depicted in Fig. 8.13. A new version of system component J will result in a new version of E, which in turn will result in a new version of B, and finally a new version of A. This ripple of version changes is in many cases an undesirable property of the hierarchy which leads to difficulties in handling temporary inconsistencies. A number of small modifications in components deep down in the hierarchy will result in a large number of version changes for components higher up in the hierarchy. Thus, automatic creation of a new version of a component whenever there is a version change in a subordinate component is undesirable. We propose to let system components be related to their decompositions by a **solved by** relationship to avoid this cascading of changes of versions.

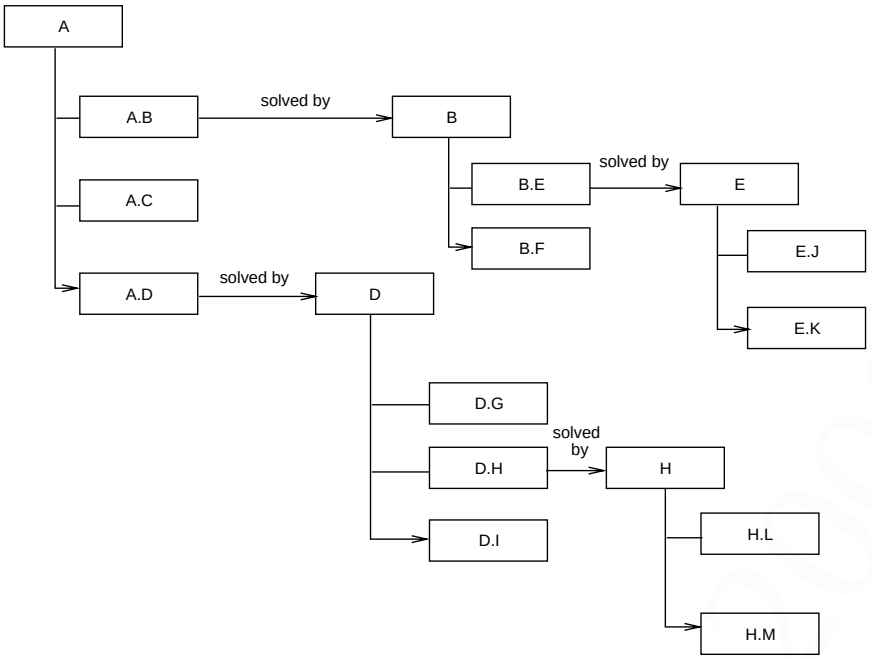


Fig. 8.14. Component structure graph

If we use the solved by mechanism it is possible to delay modifications to a component (i.e. new versions,) to take effect on components higher up in the hierarchy until they are explicitly included. Thus, new versions of components are created explicitly. Once a new version of a component becomes available, it is not mandatory to include it in any superior component. The inclusion of any new versions of subordinate components is done explicitly. Hence, inconsistencies among the development objects are tolerated. We propose the component structure graphs to represent system structure, where each component may exist in a number of versions, and the solved by relation represent the relation between a system component and its subordinate components. Figure 8.14 depicts the general hierarchy from Fig. 8.13 as a component structure graph.

There are four possible operations on hierarchical components: New hierarchical, modify hierarchical, remove hierarchical, and merge hierarchical.

- New hierarchical defines a new hierarchical component from scratch. The new hierarchical operation defines a placeholder in a new version graph for the new hierarchical system component. The default initial version is 1.0.
- Modify hierarchical defines a new version of a hierarchical component. The old version of the hierarchical component is left unchanged. The new version may be either a revision or a variant of the previous version.

- Remove hierarchical deletes the last version of a hierarchical component. Versions of a hierarchical component within a revision chain cannot be removed. The remove hierarchical operation does not affect any other versions of the hierarchical component.
- Merge hierarchical merges two hierarchical components into a single component. The merge hierarchical operation takes as input two different development versions of a hierarchical component belonging to two different development transactions. The operation creates a new hierarchical component which contains the result of the merging operation. The merging operation may be entirely manual or semi-automatic.

When hierarchical components are merged, names of the sub-components existing in the different hierarchical components are compared. Names of sub-components in the hierarchical components that are being compared, the input components, are left unchanged. In the case of a conflict of names among the input components in the merging process, the sub-component in question will have to change its name in the merged component. Name conflicts will in general have to be resolved manually. We will look into merging mechanisms in more detail below.

Flat component structures. Flat models, e.g., traditional Entity Relationship diagrams, have no inherent hierarchical structure. Components of varying significance and multiple levels of detail are logically placed on the same level of abstraction, i.e. in the same diagram. There is usually no decomposition mechanism. Notable exceptions exist, such as Tempora ERT. However, this does not mean that decomposition in the sense used for hierarchical models exists in flat models. Generally, every system component that is of the flat model type is atomic. Every system component of the flat type will nevertheless be subject to development and subsequent modifications and thus exist in a number of versions. Although system components of the flat type are considered atomic from a system structure perspective they are not atomic from a systems development perspective. E.g., a process in a Data Flow diagram will only reference certain entities, attributes, and relationships in an Entity Relationship diagram. A subdivision of tasks among several developers based on the hierarchies in for example Data Flow diagrams, will thus have as a consequence that all of the tasks reference, and may want to modify, the same flat component, Consequently, there is a need for developers to be able to reference, enhance, and modify only parts of a flat model, although this flat model is common to a large number of developers.

We propose to use scenarios to impose a structure on top of flat components. Different application projects will develop different scenarios based on differences in application needs and perception of real world phenomena. Hence, the scenarios constitute a classification of real world features according to the needs and perceptions of the applications. The scenarios may therefore be used to facilitate co-existence and integration of discourse systems developed by different application projects. The union of the scenarios constitute

the complete conceptual model. More generally, one can use the notion of conceptual views applied to all flat models. In our context the definition of a conceptual view becomes:

A conceptual view is a logically connected subset of a flat model.

Conceptual views may overlap, such that the same development component may exist in several conceptual views. There are no live links between components occurring in several conceptual views, e.g., updates are not automatically propagated among conceptual views. Conceptual views may exist in several versions. As mentioned above, one of the prime purposes of introducing the conceptual view is to facilitate co-existence of discourse systems. This means that conceptual views are related, and thus form a structure. A conceptual view can be built by selecting components from an existing conceptual view. When a conceptual view is built by selecting components from an existing conceptual view, the relationship, *built from*, between the existing and the new conceptual view is recorded. The default initial version of the new conceptual view thus created is 1.0. The built from relationship may be applied on all conceptual views, thus creating hierarchies of conceptual views built from conceptual views built from conceptual views, etc.. There are four possible operations on a conceptual view: New conceptual view, modify conceptual view, remove conceptual view, and merge conceptual view.

- New conceptual view defines a new conceptual view from scratch. The new conceptual view operation defines a placeholder in a new version graph for the new conceptual view. When a conceptual view is built from scratch, no indication of origin is given. The default initial version of a conceptual view is 1.0.
- Modify conceptual view defines a new version of a conceptual view. The old version of the conceptual view is left unchanged. The new version may be either a revision or a variant of the previous version. The successions of versions are handled by the version graphs introduced in the previous section. Variants of a conceptual view would be used to capture situations where it is desirable that different versions of the same conceptual view coexist over time. For example, an Entity Relationship diagram that exists in two versions that are variant related, to serve as the basis for two variant related data flow diagrams. An alternative way of realizing parallel versions is to build a new conceptual view based on an existing version. This would establish the new conceptual view as built from related to the original conceptual view, and not as a variant to its origin. The differences are few. But, rather than restricting the use of the variant graphs applied to conceptual views to simple revision chains, we have decided to keep the redundancy and leave the choice between the two alternatives to the preferences of user.
- Remove conceptual view deletes the last version of a conceptual view. Versions of conceptual views within a revision chain cannot be removed. The

remove conceptual view operation does not affect any other conceptual views or any other versions of the conceptual view.
- Merge conceptual view merges several conceptual views into one single conceptual view. The merge conceptual view operation retains the old conceptual views, and creates a new conceptual view which contains the result of the merging operation. The merging operation may be manual or semi-automatic.

When conceptual views are merged, names of phenomena existing in the different conceptual views are compared. In general, name conflicts will have to be resolved manually.

In the general case, manual intervention to negotiate the inconsistencies that have developed among the input conceptual views is required. It is generally not possible to automatically determine any optimum conceptual view to which the merged conceptual view by default would be related. The definition of optimum is subject to personal interpretation and the intention of the merge operation rather than the origin and history of the input conceptual views.

It may be the case that the result of the merge operation is such a major restructuring of either of the input conceptual views, that the feeling of the teams is that it would be counter-productive to continue to use the input conceptual views as the basis for further development. This may be particularly so knowing that a new merge operation inevitably is going to take place at some future point in time. In this case, an alternative solution is to build two new conceptual views based on the reconciled version, and use these new conceptual views as the basis for further work.

Global Name Uniqueness. It must be possible to refer to all development objects in an information system by a unique name. We will present a naming schema for development objects that provide and maintain global name uniqueness. There are two alternatives for obtaining global name uniqueness:

- A flat namespace where the name uniquely identifies the development object no matter where it is being used.
- A hierarchical namespace where development objects names are qualified with the names of hierarchically superior development objects.

The simplest solution comprises a flat namespace where a development object's name uniquely identifies the development object independently of where it is used. For small systems this is an ideal solution. Each development object has its own name distinct from any other development object. Referring to a development object by name is simple and without ambiguity. However, as the number of development objects grows a flat namespace will quickly become impossible to manage. A solution where development objects' names are qualified with the names of hierarchically superior development objects remedies the problem of namespaces growing out of bounds. A hierarchical qualification system allows a flat namespace of moderate size

to be seen as a hierarchy. We propose to have a hierarchy of project libraries. Each development object should be uniquely named within each library. Development objects within a development object should be uniquely named within that development object. Global uniqueness is accordingly ensured through qualification of a development object's name with the name of the superior development object.

Hierarchical models. In the case of hierarchical models, the name uniqueness requirement imply that each development object will have to be uniquely named within each development object for each library. Names of components are qualified by the name of the ancestor development objects. Thus, in the case of e.g., data flow diagrams, the name of each component in a diagram will have to be unique. Each process, data store, data flow, and external entity is uniquely identified by its name within a diagram.

Flat models. Flat specification models inherently have a global name space. The conceptual view concept superimposes a structure on the models. The structure is motivated purely by pragmatic considerations within the development organization, e.g., the split of tasks among developers. The name uniqueness requirement applied to conceptual views imply that components' names will have to be unique within a conceptual view and not within the complete conceptual model for a domain. Names in different conceptual views are not necessarily unique. Names of components are qualified by conceptual view name. E.g., in the case of a conceptual view defined over an Entity Relationship diagram, each component in the conceptual view will have to be unique. Each entity, relation, and attribute, is uniquely identified by its name within the conceptual view.

Granularity of Versioned Components. To be able to keep concurrent versions of system components, we must determine a suitable level of granularity of atomic system components. The granularity will have to be determined by the grouping of selected basic modeling constructs into appropriate atomic components. The atomic system components will subsequently be composed into structural components which will form a hierarchy of system components. The decision on granularity will reflect a compromise between the high complexity and high costs that are associated with a fine grained solution, and the relative loss of control and flexibility that come with a coarse grained solution.

Hierarchical models. As a representative of hierarchical models, we consider the modeling constructs of Data Flow diagrams. The Data Flow diagrams comprise a natural hierarchy through the decomposition of processes. Revision of a process happens through a revision of its decomposition. Other components of Data Flow diagrams are data stores, data flows and external entities. Those components are not refined through hierarchical decomposition as is the case for processes. From a version handling perspective, refining a data store or a data flow is equivalent to deleting the old data store or data

flow and inserting a new. Thus, the process component of a Data Flow diagram is the lowest level development object that will exist in several versions.

Flat models. Regarding flat models, e.g., Entity Relationship diagrams, the feasibility of keeping versions of entities, relationships, and attributes is in our opinion questionable. The feasibility of keeping versions of conceptual views, i.e. entire diagrams of some manageable size, is much more obvious.

Selecting a Version. A specific version of a development object may be selected and accessed by browsing of the component graph in combination with simultaneous browsing of the version graph.

- Having chosen one specific development object version in the component structure graph it is possible to switch to the corresponding version graph to see the complete version picture for that development object. The last version worked with will be the default.
- Having chosen one specific version of the development object in the version graph it is possible to switch to the corresponding component structure graph to display its subordinate components. If this is an atomic component one may switch to browsing of its content.
- Having chosen one specific version of a development object in the version graph it is possible to start the development of a new version of the development object based on the selected version. The new version can be related as serial or parallel to its origin.

8.3.2 Way of Working

This section briefly describes a development conceptual view which serves as an illustration of the use of the concepts and functionality presented in the next two sections. The section describes the application of this in PPP.

Selecting a version of a development object. Selecting version 1.2 of Address register, is done by marking it in the version graph, and choosing browsing or check-out from a pop-up menu. Browsing would result in the creation of a read-only window displaying the contents of the development object using the appropriate tool or editor. Check-out would result in the creation of a placeholder in the version graph. The placeholder will have to be placed either as a revision or as a variant to the version that was checked out.

Development transactions. When a version of a development object is checked out for further development, the version of the object is copied to a private workspace and may be modified by the developer using a set of appropriate tools. Elapsed time before check-in may be in the order of days and weeks. When the modifications are completed the object is checked in to the specification base. The task of developing a new version of a development object , i.e. from check-out till check-in, is termed a development transaction. As development transactions may be arbitrarily long lived, facilities for sharing

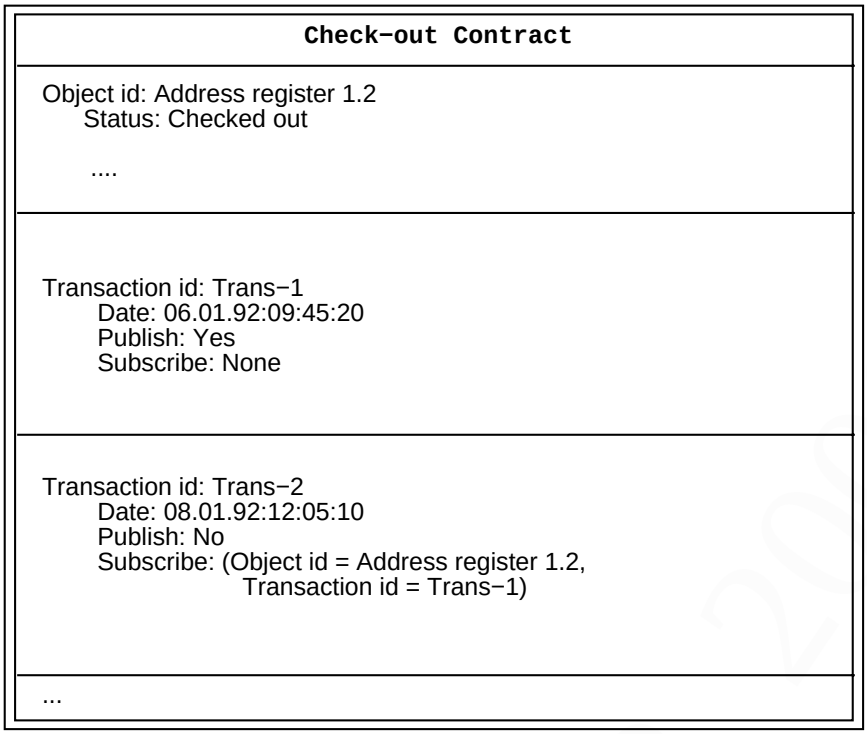


Fig. 8.15. Check-out contract for Address register 1.3

of specifications between transactions are provided. Several transactions are permitted to concurrently access and check-out the same version of a development object. Thus several copies of the same version may be in the process of being modified simultaneously by different transactions. Modifications made to the development object by the different transactions are by default considered private and kept separate until check-in time. Synchronization among transactions is provided by the publish/subscribe mechanism that allows one transaction to subscribe to local revisions of development objects developed and published by other transactions. Check-in time is determined by the developer responsible for the development object. Suppose that two different developers were interested in modifying Address register 1.2 with the intention of creating a new version. Both developers would check out the development object in separate development transactions, say Trans-1 and Trans-2. For each transaction initiated a transaction log is created. The transaction log is owned by a transaction, and contains information on the development objects currently checked out by the transaction. When the first transaction checks out the development object, a placeholder for the new version is created and a check-out contract for the new version of the object (the placeholder) is created. The check-out contract will be updated with information on date

and time of check-out, transaction identifier, whether local revisions created during the transaction are to be published or not, and subscriptions to local revisions of development objects created by other transactions. Let us assume that the first transaction checking out version 1.2 of Address register places the new placeholder as a revision of version 1.2. The placeholder will thus be named Address register 1.3. The check-out contract for Address register 1.3, after transactions Trans-1 and Trans-2 have checked out the object is depicted in Fig. 8.15.

Check-in of a new version of a development object. At check-in time the development object is returned to the specification database with the purpose of updating the database with a new version of the development object. If there are several candidates for the new version, i.e. if the development object was checked out by several transactions and these transactions have modified the object, the candidate versions must be merged. All developers having a copy of the development object are notified, and a suitable time to attempt the merging is negotiated. Some developers may need an extra day or an extra week before being ready to merge with the others. In the end, the time of merging will have to be determined by the developer formally responsible for the development object. When the merging of the candidate versions is attempted, conflicts among the candidate versions may be detected, and a cooperative session among the developers of the candidate versions will have to be initiated. The purpose of the cooperative session is to negotiate a new version of the development object. The negotiations are based on the conflicting candidates. The session may be synchronous or asynchronous. Tool support is provided through multi-user editing capabilities. In the synchronous case all candidate versions are available to the involved parties during the session through a multi window interface. A developer may comment all proposals. Various candidates may be compared, both on a non-executable (textual or diagrammatic) and an executable (by simulation or execution) basis. Cut and paste among windows to achieve a reconciled version is supported. In the asynchronous case one have to rely on written communication in the form of comments added, and cut and paste versions, to convey opinions among the developers involved in the negotiation process. A successful negotiation session results in an agreement on one reconciled version that will be the next version of the development object in question. This version will be stored in the specification database, and the development transaction is terminated. Suppose that development transaction Trans-1 is ready to check-in the development object at Date = 13.01.92 : 15:05:50. Additionally we assume that the developer responsible for transaction Trans-1 is also responsible for Address register. This means that in the end it is the owner of Trans-1 who determines the time of check-in. When Trans-1 issues the request for check-in, the check-out contract for the development object is inspected. All transactions that is registered as having checked out the development object is notified of the request for check-in. Suppose that transaction Trans-2 requests a one

week postponement before merging, the owner of Trans-1, being the developer responsible for the development object, is at liberty to grant or reject the request. When merging potentially is needed, i.e. when several candidates for the next version of the development object exists, there are two possible outcomes:

- The developer responsible for the object picks one of the candidates. The chosen candidate will be checked back into the repository as the next version of the development object.
- The developer responsible decides to merge the candidates. There are two ways of attempting to merge the candidates:
  - Try an automatic merging of the candidates. If overlap is detected, initiate a cooperative session to negotiate a new version.
  - Initiate a cooperative session to negotiate a new version directly without trying to merge automatically first.

When the merging process is finished, the result is checked back into the specification database as Address register 1.3, i.e. filling the placeholder for this version of the development object.

8.4 Use of Viewspec in Modeling

From the models, one can generate a set of viewspecs which we would like to be able to work with simultaneously, i.e., we want to allow *multiple viewspecs to co-exist*. To use multiple viewspecs effectively in the modeling process requires that:

- *Management* of multiple viewspecs is supported.
- It is possible to go *back and forth between models at different abstraction levels*, i.e., flexible ways to go back and forth between full models and associated viewspecs.

Management of multiple viewspecs. To allow multiple viewspecs and multiple viewspecs to co-exist, we distinguish between three different relations in the versioning graph: filter, variant and context relations.

A filter relation depicts the relationship between a viewspec and its originating model. Two models are filter related if one model is generated from the other model by a filter. Thus, a filter relation depicts relationship between a full model and a viewspec or between two viewspecs. Moreover, a model can be filter related to *several* viewspecs, that is, several viewspecs are generated from the same model. Figure 8.16 shows a versioning graph that depicts a situation where a transaction has checked out a model *S1.1* and a set of viewspecs have been generated from it. A model is uniquely identified by name $S1.1\{\tau\}\langle\alpha\rangle.\lambda$, where $\{\tau\}$ indicates it belongs to transaction τ, $\langle\alpha\rangle$ indicates that it is a viewspec of type α and λ is the revision number local to the transaction.

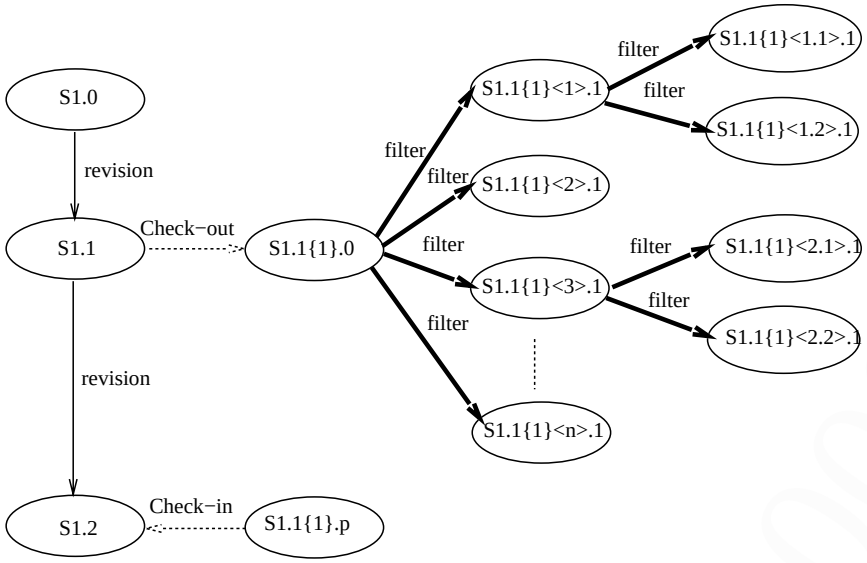


Fig. 8.16. Viewspecs are filter related to their originating models

Figures 8.17a and b shows a version graphs for a data and process model, respectively.

We make a distinction between updates of a viewspec that concern the contents, i.e., change of semantics, and updates that concern the representation of a viewspec without changing the semantics, e.g., restructuring of diagrams. In a similar manner as for full models, it is useful to talk about *revisions* and *variants* of viewspecs. An updated viewspec is a revision of its originating viewspec, whereas a viewspec is a variant of another viewspec if it is expressed using a different language or its layout is modified but in both cases the semantics of the models remain unchanged. We postpone the discussion of updates of viewspecs until the next section. A variant of a viewspec is denoted by a variant relation. Accordingly, we are allowed to keep different variants of the layout of a viewspec. It is up to the analyst to decide how many layout versions of a viewspec she actually wants to keep. If we want to include the intermediate viewspec that is the viewspec which is the result of the filter before modification of layout, we indicate this in the versioning graph by relating the viewspec and the restructured viewspec by a variant relation (Figure 8.18). The L indicates that the version is the result of changing the layout of the viewspec without changing its semantics.

When working with a combined language like PPP, it is useful to be able to have an easy access to related viewspecs expressed using a different language. This is provided by introducing the context relation. A viewspec V1 is context related to another viewspec V2 if V2 describes related aspects of V1, i.e., V2 elaborates the context modeled in V1. The viewspecs are closely

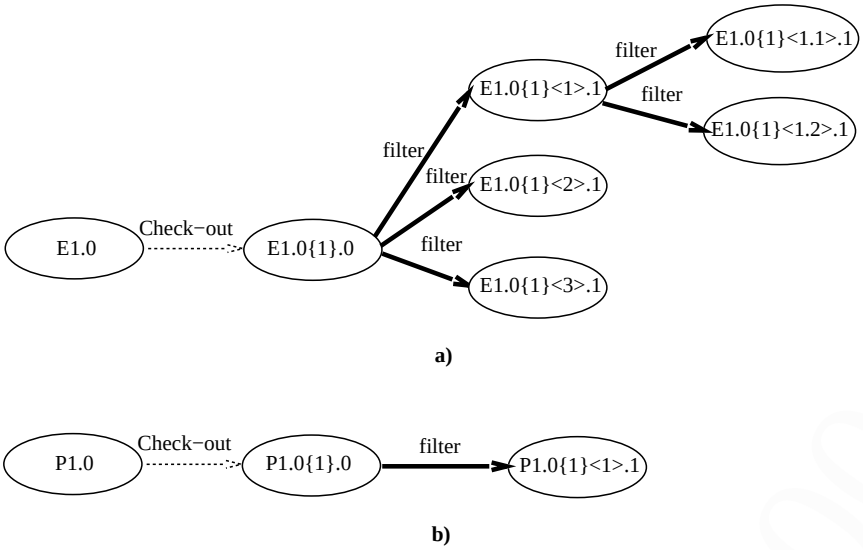


Fig. 8.17. a) A data model and associated viewspecs b) a process model and associated viewspecs

related and the developer may go back and forth between the viewspecs. To support this way of working we therefore relate the viewspecs by a context relation in the versioning graph (Figure 8.19). $P1.0\{1\}\langle 1\rangle.1$ corresponds to the viewspec depicted in the version graph in Fig. 8.17a.

Back and forth between models at different abstraction levels. It should not only be possible to allow multiple viewspecs to co-exist but it should also be possible to go back and forth between models at different abstraction levels in a flexible manner. This implies that it should be possible to go back and forth between a full model and its associated viewspecs as well as different versions of such models. To keep track of all the models we have already introduced some relations for depicting relationships between models. In addition, we introduce the notion of *local workspaces*. The purpose of local workspaces is to *hide irrelevant models*, provide easy access to relevant models and provide a more comprehensible naming schema for relevant models. Practical usage of the viewspecs have shown that the number of viewspecs associated with a large model may become large and its difficult to keep track of all of them.

Although the definition of transaction can be considered as a local workspace, the introduction of a flexible filtering mechanism calls for a finer granularity of workspaces than is provided by development transactions. Thus, we suggest that a transaction can be divided into a number of local workspaces. These are defined by the developer according to her needs. Each local viewspec may consist of a set of full models and a set of associated viewspecs, i.e., a subset of a development transaction. Figure 8.20

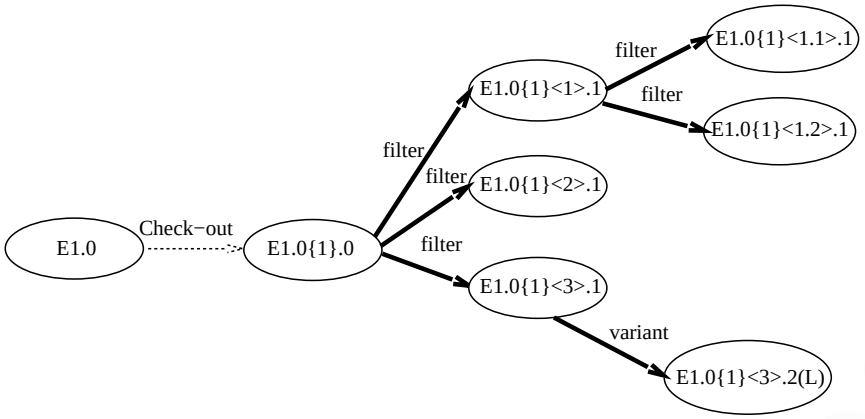


Fig. 8.18. Viewspecs are variant related to their originating models

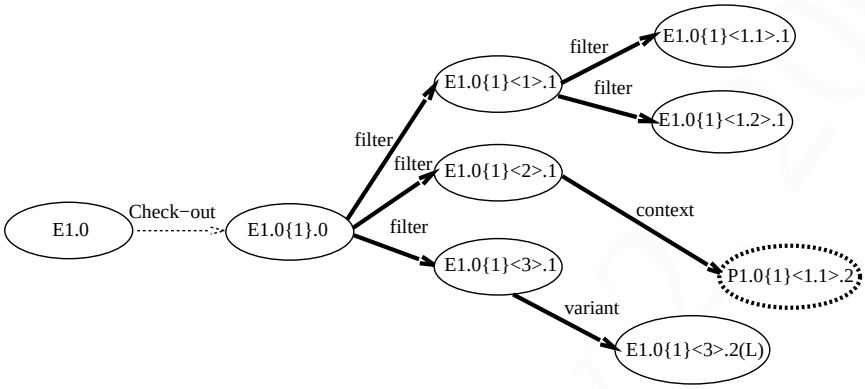


Fig. 8.19. Viewspecs are context related to their originating models

shows an example of how local workspaces can be denoted in a versioning graph. The selection of viewspecs to participate in a particular workspace is a user-decision.

The advantages of providing the user with the ability to define her own workspaces are the following:

- Easy to keep an overview of relevant models for the task to be undertaken at any time during the development process. The relevant viewspecs can be shown by a *graphical facility* which provides overview pictures of global and local workspaces. Such a facility would utilize the information already provided in the versioning system. The use of local workspaces may resolve the conflict of providing a sensible context for carrying out the actual work. Thus, the concept of complexity reduction also applies to versioning.
- Easy to go back and forth between different abstraction levels. Advanced user interface packages facilitate such a working mode by allowing several

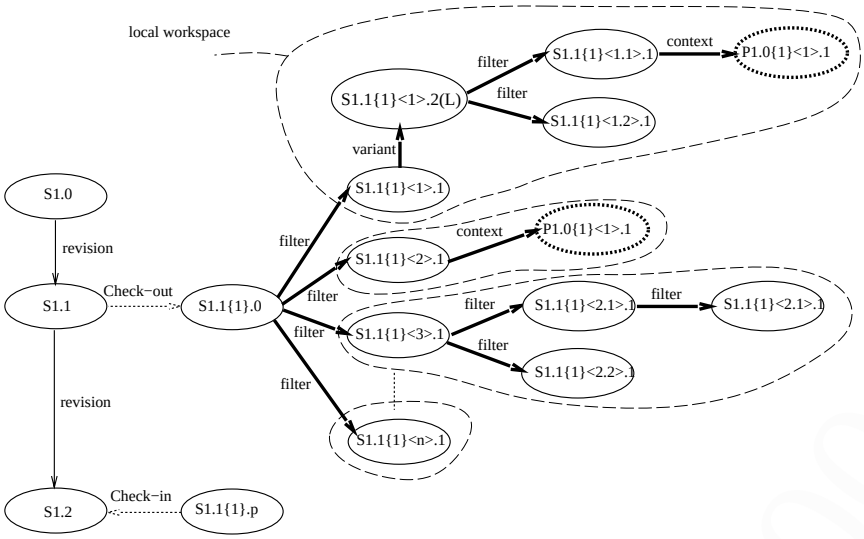


Fig. 8.20. Local workspaces

windows to be displayed at the same time, cut and paste between windows, *etc.*

– The introduction of local workspaces allows us to have local name spaces for viewspecs. A more flexible naming schema is necessary. One should not be disturbed by transaction name and full model name. The name of a version of a viewspec must be unique within the local workspace but different workspaces may contain common names. A component of a local workspace can be uniquely identified by prefixing the component name by the unique name of the checked-out originating model and the local workspace name. Thus, the system should work with surrogates internally which uniquely identify models. However, the developer only deals with simpler name within each local workspace.

Some of these ideas can also be extended to encompass versioning of full models. However, we will not pursue this here.

8.4.1 Inserting Modeling Statements

This section details how viewspecs can be used as a basis for further modeling.

Why Allow Updates of Viewspecs?. By allowing updates on viewspecs we may use them as a basis for *entering new statements* into models as well as for presentation purposes. The rationale behind allowing viewspecs to be updated is:

– *Large and complex models are difficult to update.* Rather than entering new information to a large and complex model, the developer can concentrate

on relevant details by updating appropriate viewspecs generated from a full model. The developer is not confused by irrelevant details and thus, modifying a smaller and less complex model requires less effort.
- *Explore alternatives.* In some situations, the actors envisage several alternative solutions which may be interesting and important to pursue. The use of viewspecs allows different solutions to be modeled and it also provides support in managing the resulting models.
- *Conflicting views of reality are useful to model as intermediate results.* Different parts of a model may be based on slightly different and inconsistent views of the reality. To resolve potential conflicts among these views, it may be useful to model these as intermediate results. In some way, the final information system must accommodate a compromise of all these views.
- *Generate appropriate viewspecs for presentation.* We have defined a set of filters. The user may however want to generate a viewspec for which no filter is defined, e.g., the resulting viewspec may still contain irrelevant information which must be suppressed manually.

The importance of allowing viewpoints to be updated has also been pointed out by others, e.g., Easterbrook [100].

Management of Updated Viewspecs. We have mentioned before that revisions of full models has a counterpart in updated viewspecs. Figure 8.21 shows a version graph for a full model with associated viewspecs. Viewspecs are updated and this is depicted by a revision relation. It is the contents of the viewspec that are changed, e.g., by adding or deleting information. Each update of a viewspec results in a new revision of the viewspec.

The Problem of Inconsistency. Allowing viewspecs to be updated implies that they can be modified *directly* by the developer. It is up to the developer to manipulate the viewspec and therefore, no control is to be provided by the system beyond consistency checking which is also performed for full models. However, by allowing updates on viewspecs another type of inconsistency problem also arises. Updating a viewspec may imply that the updated viewspec and the full model from which the viewspec originates, become *inconsistent*. The inconsistency may occur for several reasons. Updated viewspecs may have (relative to the originating model from which they were generated):

- New components.
- Updated components (slightly modified components).
- Deleted components.

Conflicts may occur such as conflicting names, conflicting types and conflicting use of domain concepts in different models. Also conflicting views of the reality can be explicitly modeled.

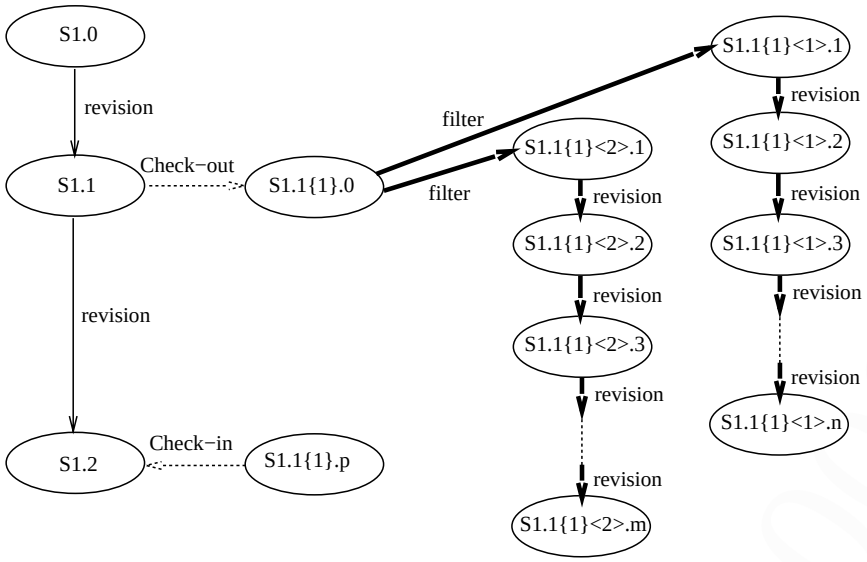


Fig. 8.21. Revisions of models

Alternative Ways of Dealing with Inconsistency. In spite of the problems of inconsistencies, we have chosen to allow updates on viewspecs because we want the viewspecs to be used 'actively' during the development process. It is crucial that the developer can enter new model details into viewspecs. The developers can thus concentrate on relevant issues without being disturbed by irrelevant details which is essential when dealing with large and complex models. The price we have to pay for allowing updates of viewspecs is to provide *support* to assist the developer in dealing with inconsistent models. We envisage at least three solutions:

1. Allow updates of viewspecs but insist on that inconsistencies should be resolved before any other action can be taken. This could be done by creating a skeleton, i.e., a *placeholder*, for a new revision of the full model as soon as a viewspec is updated (Figure 8.22). The conflicts between the viewspecs and the originating model must then be resolved before the development can proceed.
2. Allow updates of viewspecs and allow for inconsistencies to exist but provide extensive support for the eventual model merging process (Figure 8.23).
3. Allow updates of viewspecs and allow for inconsistencies to exist but leave the model merging process to the developer.

The first alternative corresponds to a rigid development approach, where it would not be allowed to have several updated viewspecs to co-exist. This contradicts our requirement to a flexible approach. If we go for the third

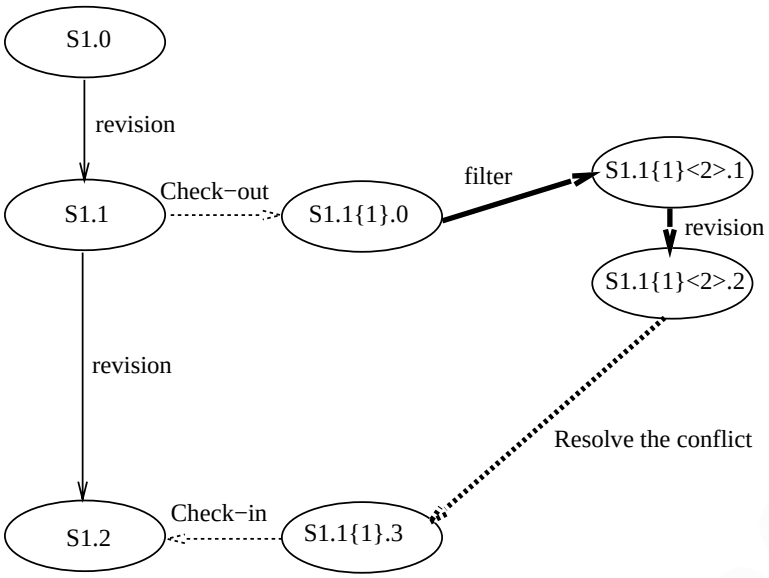


Fig. 8.22. Resolving the conflict immediately after a viewspec is updated

approach we only need to provide versioning of updated viewspecs and leave the process of integrating the new versions into the full model to the developer. This does not seem to be the way to go due to the fact that providing this flexibility may do more harm than good in the development process. In particular, this alternative may become quite time-consuming when many viewspecs are updated and used as a basis for updating a full model. We have to come up with facilities to support the process of going from one full model to another one by taking into account updated versions of viewspecs. Thus, the second solution is what we will go for in the remainder of this chapter. The next section outlines support for the process of building full models based on a set of updated viewspecs.

8.4.2 Inclusion of Changes

This section details how changes that are made to one or more viewspecs can be used as a basis for building a revised version of a full model. To carry out the process of including the changes contained in the viewspecs into the full model, is not straightforward. During the modeling process several viewspecs may be updated and *several* versions of each viewspec may *co-exist*. Thus, an important step in a modeling process based on viewspecs is to provide support for the process of solving eventual *conflicts* between updated viewspecs and the originating model. We distinguish between two major ways of including changes:

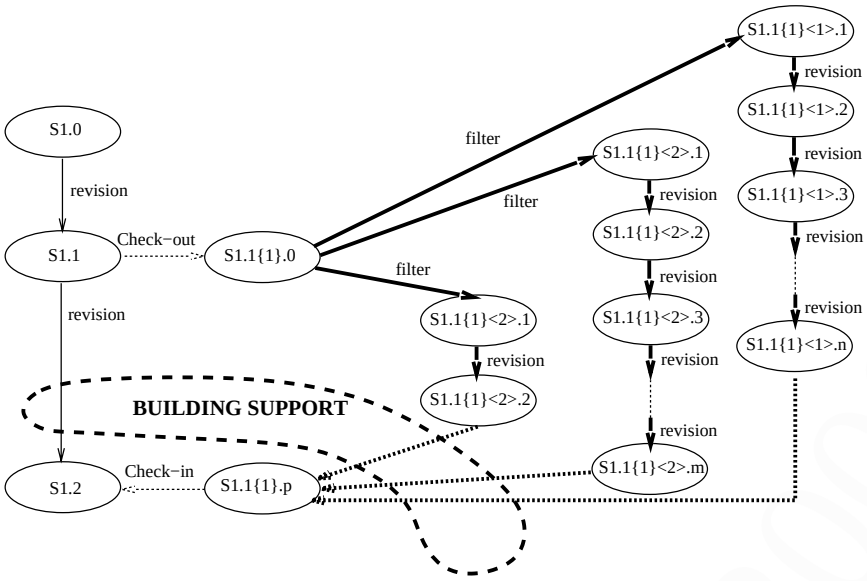


Fig. 8.23. Building a new revision based on updated viewspecs

– Change propagation.
– Model integration.

Controlled Change Propagation. The simplest way of including changes on updated viewspecs into the respective full models is to propagate the changes one by one. Changes to a model are *not* by default propagated automatically. For large models the number of interconnections is usually very high and the consequences of the changes may not be predictable, e.g., an update of an entity class in the ONER-model may cause several changes to DRL rules that contain ONER expressions that refer to the entity class. To propagate changes in a controlled manner we advocate an approach based on a mixture of automated support and manual intervention. In general, changes to a viewspec V are only considered for propagation if the viewspec is selected by the developer. For example, if a viewspec is updated and only used for presentation purposes, we do not consider to propagate the changes.

An updated viewspec may contain modified objects that should also be propagated to models other than the originating model. By using the links that define the interrelationships between the languages, a list of affected objects in associated models can be provided. Thus, search facilities that utilize intra-language, inter-language and inter-level links are valuable support when changing/modifying a model.

Warnings could also be provided to inform the developer that other associated concepts are affected and must be updated in order to maintain consistency across model boundaries. Then the developer may choose whether

she wants to propagate a change or not. The changes are carried out one by one and some may also be propagated automatically.

In some cases, automatic propagation of changes may be appropriate but it should be user-driven and only performed if one can assure that it has no undesired side-effects. That is some on-off mechanism should be provided. This can be illustrated with a simple example. Let us say that we want to change the name of an entity class from adr to address. It is obvious that before the change is propagated the developer should be provided a warning if address already exists in the models. If address does not exist the change can be propagated to all models, going across model boundaries using appropriate links.

Model Integration. The process of putting a new revision of a full model together based on viewspecs will be referred to as the *model integration* process. The result of the model integration process is a *consistent version* of the previous full model including the changes in the selected viewspecs.

A thorough analysis of model integration is beyond the scope of this work. We limit ourself to describe major steps of the integration process and to outline some means to support such a process. The support itself will be a number of very specialized tasks as outlined in Chap. 7.

The input to the model integration process. Figure 8.24 shows how models that form the basis for a model integration process are depicted in a versioning graph. The input to the integration process is related with *merge* relations between the relevant models. Possible models are:

- A previous version of the full model.
- One or more update viewspecs.
- One or more viewspecs which are approximations of the corresponding originating model.

Additional information may be provided about each updated viewspec, e.g., type and what changes have been made since it was created. This information is recorded during the filtering and updating processes. The versioning facility provides the information together with the names of the models. The models are then retrieved from the model repository.

Only viewspecs which contain approximations of the originating model and updated viewspecs should be considered for integration. Other viewspecs are not relevant because they are projections of the originating model and thus, the viewspec's information is already included in the full model. However, not all updated viewspecs are considered for integration. It is a *user decision* to select what viewspecs that should be considered for integration. For example, if a viewspec is updated and only used for presentation purposes, we do not consider to propagate the changes. We may also have situations where updated viewspecs are used to explore alternative solutions/conflicting viewpoints. Only the set of viewspecs that represent the 'consensus' solution

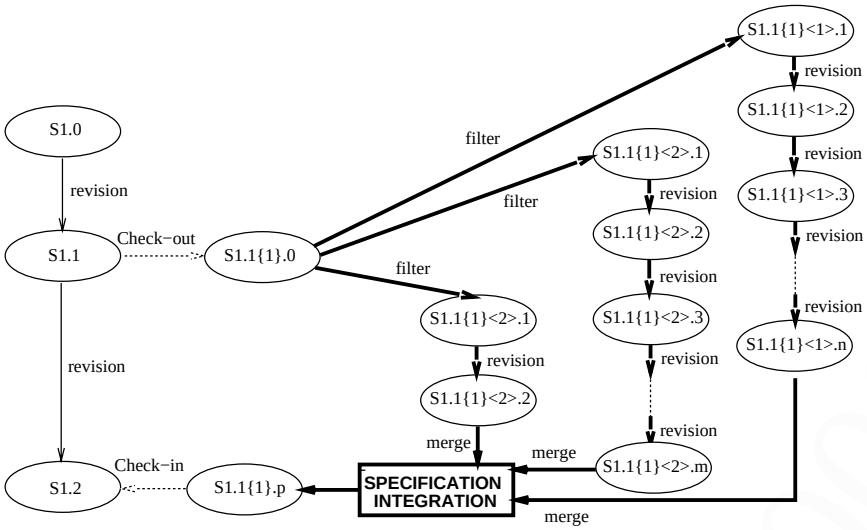


Fig. 8.24. The merge relations

are considered for the integration process. In this case, we envisage that additional facilities are provided to support the process of arriving at a feasibly agreed solution.

The viewspecs used as a basis for the model integration may contain (relatively to the full model from which they originate): new components, updated components, deleted components, and/or redundant components. If an update of a viewspec only results in redundant components the updated viewspec does not need to be considered any more in the integration process. If components are deleted from the viewspec, corresponding components in the full model must be considered for deletion. If new components are added and they do not exist in the model, they must be considered for insertion in the new version of the model. Updated components implies that the corresponding components in the full model must be considered for modification. The detailed model integration process was discussed in Chapter 7 and is not repeated here.

8.5 Chapter Summary

The methodology outlined in this chapter can be given the following classification when using the framework presented in Sect. 8.1:

– Weltanschauung: Constructivistic. This view has been important for the specification of guidelines for a more detailed methodology.
– Coverage in process: Both development and maintenance are supposed to be covered in an integrated manner. Usage and support is also ad-

dressed [209] although not in the same detail. Planning and management issues are neither covered in detail. It is basically aspects that are related to conceptual modeling which have been described in much depth.
- Coverage in product: The whole portfolio is meant to be supported, although most of the suggestions in this thesis are geared towards the support of single application systems. The set of modeling languages also makes it easy to model aspects of strategies, business processes, and organizational structures and interrelationships between people in the organization.
- Reuse of product and process: Generative reuse is supported. Suggestions for supporting compositional reuse have been made, although much work is still to be done on this area.
- Representation of product and process: The methodological framework is based on active use of conceptual models written in languages covering several perspectives to modeling.
- Participation: Joint decision making guiding the development.
- Maturity: Low.

Looking more generally on the work presented in literature, we can conclude the following:

- Weltanschauung: As also noted by Hirschheim and Klein [170], most earlier and current methodologies for application systems development and maintenance have an objectivistic outlook. Some exceptions are STEPS [120], Multiview [16], and our own methodology [207]. Other examples are methodologies based on SSM [61] and some PD-methodologies [325].
- Coverage in process: Most methodologies for CIS-support are focused on development, with maintenance being looked upon as a separate endphase. Several methodologies focused specifically on maintenance also exist (e.g.CONFORM [52], see also [34]), even if this part of CIS-support is not shown the same interest as development by researchers according to [159, 194]. Some methodologies covers both development and maintenance in the same framework in an integrated manner (e.g. The Spiral Model [32], the Hierarchical Spiral Model [176, 177] and the framework presented by Basili [21] where also emergency error-corrections are covered). STEPS [120] and our own methodology [207] also include aspect related to usage. METHOD/1 [9] is strong in the area of managing and planning.
- Coverage in product: We have found few methodologies apart from the one we have suggested that cover traditional development or maintenance of the whole portfolio in a focused manner, even though maintenance can be said to often be performed in this way [357]. Several methodologies include organization-wide CIS-planning (e.g. METHOD/1 [9]). Approaches to enterprise-modeling (e.g. [324]) are also in this category, but they only look into some aspects of CIS development and maintenance.
- Reuse: Some methodologies explicitly addressing reuse exist (e.g. REBOOT [199]), even if few development and maintenance methodologies are

geared towards conscious component reuse. Operational and transformational approaches as described in [408] are highly geared towards generative reuse. This is also the case with Tempora and our own methodology.
- Use of conceptual models: Many methods use conceptual modeling to some extent, even if most use only semi-formal modeling languages e.g. most OOA and OOD-approaches. On the other hand, the use of operational conceptual models have received increased interest as illustrated through Tempora and our own methodology.
- Stakeholder participation: Increasingly looked upon as important both in objectivistic and especially constructivistic methodologies. This might be endangered by the current trend of more and more use of packages and outsourcing, although this might have other advantages.
- Maturity: Most mature methodologies resembles the traditional waterfall model, but many of these, for instance Method/1, are taking newer aspects into account. Most methodological frameworks described in literature have a very low maturity, being described academically, but not used extensively in practice. This especially applies to the methodology that we have proposed, which is the framework which otherwise are meant to best cover the other six aspects.

There seems to be an overall view that there is no right methodology for all situation [16, 120, 139, 176] something which is also recognized in more traditional methodologies like Method/1. The different development and maintenance efforts can vary according to several factors e.g.:

- The wickedness of the problem (cf. Rittel [314]).
- The complexity of the application system (cf. Brooks [41]).
- The current state of flux.
- The size, perceived importance, and risks of performing the changes (cf. Work on the spiral model by Boehm [32].)
- The number of stakeholders affected, skills needed, and possessed.
- The number of different views of the situation (cf. social construction theory as described in the introduction).

Thus there is a need for flexibility, but in our opinion one still need a methodology or a methodological framework of some sort to be able to deliver CIS-support in an organization. Taking into account the multitude of techniques, there is an obvious need for an integrative framework that can incorporate existing more detailed approaches and support their flexible situation-dependent use.

A. Evaluating OMT Using the Quality Framework

We will in this appendix illustrate how the quality framework can be used in evaluating existing languages and tools for conceptual modeling.

We have chosen OMT (Object Modeling Technique) [319] and the tool support given for it in StPs OMT-tool as our example for showing how the framework can be used for evaluation purposes. OMT is a well known and well established approach. Place limitations hinders us to also assess additional approaches. Specifically, an evaluation of UML would be interesting to do to compare it with OMT.

OMT is briefly described in Sect. 2.2.6, and we will only recap the main aspects here.

OMT has three modeling languages: the object modeling language, the dynamic modeling language, and the functional modeling language. The Object modeling language describes the static structure of the objects and their relationships. It is basically a semantic data modeling language with some extensions for the modeling of instances (i.e. objects). The dynamic modeling language, which is based on Statecharts [162], describes the state transitions of the system being modeled. It consist of a set of concurrent state transition diagrams. The functional modeling language describes the transformations of data values within a system. It is described using data flow diagrams. The notation used is similar to traditional DFD with the exception of the possibility of sending control flows between processes, which are signals only. Actors correspond to objects as sources or sinks of data.

The OMT methodology is divided into three phases; analysis, system design, and object design. The input to analysis is the problem statement and the output is a formal model that identifies the objects and their relationships, the dynamic flow of control and the transformation of data through the system.

Recently, many companies, such as IDE (StP), Cadre Technologies, Select Software Tools, Westmount Technology, and Platinum (Paradigm Plus) have released modeling tools to support OMT [292]. We have chosen to use the StP-version since this is regarded by OVUM [292] as the most comprehensive of the tools, specifically for analysis and design tasks, where OMT primarily focus.

A.1 Evaluation of StP/OMT

The evaluation is primarily based on [106, 178, 292, 396] in addition to [319]. The terms for specific language constructs are defined further in these works.

A.1.1 Language Quality

– Domain appropriateness:
OMT is claimed to be appropriate for a multitude of domains. Compared to the object-model in [396], OMT is one of the object-oriented modeling languages that has best coverage for object-oriented analysis and design. It supports the object and structural perspective well. Missing analysis features reported by Embley [106] are the lacking possibility of indicating synonyms and homonyms, generic classes, high-level classes and high-level relationship sets. It also supports the behavioral and functional perspective, but the perspectives are not clearly integrated. Aspects of functionality are identified in all modeling perspectives, operations in object modeling, actions and activities in dynamic modeling and functions in functional modeling. Even though it is outlined how to extract operations from the object models, state diagrams and data flow diagrams, the process seem to be quite intuitive and pays only minimal attention to the possible inconsistencies between the models and to their reconciliation. No meta-model of the languages was originally made. It can also be stated that both the support of the functional and behavioral perspective is better taken care of in other languages covering these perspectives (e.g. choice and selection and temporal aspects of Statecharts are not included in the dynamic modeling). Other missing analysis features according to [106] is interaction details, high-level transitions, and high-level interactions. The functional modeling language is basically traditional DFD, lacking e.g. continuos data flow, actors and roles responsible for carrying out processes, and logical relationships between the flows entering or leaving a process .
Regarding rules, these are partly supported through the possibility to state constraints related to classes in the object-model, with the restrictions this have [173] when it comes to represent rules applying to two or more objects. Exception handling is neither supported. Rule-hierarchies are not supported, and neither is speech-act modeling. Actors and roles can be modeled using objects and classes, but there are no constructs for differentiating these from other classes, with the exception of the external actor in the functional model.
Whereas static and dynamic aspects can be modeled, no explicit support of the modeling of temporal aspects is given.
According to Embley [106], there are several features in OMT which should be first used in design, and not in analysis. This includes: Values, attributes and methods, object-class templates, abstract classes, pseudo-inheritance, value-based identity, and semantic replacement. Obviously, one can use

OMT in analysis without applying these concepts, but there is no tool or methodology support for restricting the part of the language used for which task.

The languages have both general and some very specific constructs, and are composable in a similar way as traditional data, process, and state transition modeling languages, which means that the composability is somewhat restricted. E.g. in the functional model, every flow must at least connect to one process.

The languages can be flexible in precision, although the functional modeling language has the traditional lacks of DFD regarding formality.

- Participant knowledge appropriateness:
  The basis of OMT is a set of well known languages: DFD, ER, and state-transition diagrams. On the other hand the object-oriented emphasis of how these are applied might hinder the use at least initially, although less than with purer object-oriented approaches [1] It is not possible to model inconsistencies explicitly.
- Participant interpretation enhancement:
  Regarding symbol discrimination, we note the following:
  - Objects, processes, and states have very similar representation.
  - The symbol used for aggregation is similar to the symbol used for ternary relationships.

  Regarding the uniform use of symbols, we note that:
  - Cardinalities are shown in three different ways: As a set of numbers, as a filled or open dot, or as nothing (the last situation can give the impression that the model is incomplete).
  - Relationships (associations) are shown in four different ways, according to if they are binary or n-ary (n>2) and if they have attributes or not.
  - The actor symbol is used for modeling both actors and roles (as it is usually done in DFD).

  In the object model, emphasis is given to classes (and objects) through size which seems sensible since these are the main concepts of this perspective. Emphasis is also given to a special form of cardinality and overlapping membership in specialization through blackness which seam more arbitrary. Since one inserts attributes and methods in classes and objects, these will differ in size, and they might get visually complex. The positive aspect of this is that one have fewer nodes and types of nodes in the object model. Using OMT to represent the example in Fig. 3.3, one would need 3 less nodes, 3 less links, and the same number of textual annotations.

  Similarly can be said about the introduction of the union-notation for the state-diagrams, where the improved expressive economy is clearly more important than the negative aspect of non-uniform size.
- Technical actor interpretation enhancement:

[1] OMT is regarded as a hybrid SA/OO approach since it borrows quite a lot from structured analysis [396].

All modeling languages have well-defined syntax. Both the object and dynamic modeling language have well-defined semantics.

A.1.2 Potential for Creating Models of High Quality

When referring to tool-support below, we refer to what can be found in the tool made by IDE (StP).

- Physical quality: StP is faithful to OMT with two exceptions: Event hierarchies are not supported, and instance diagrams are substituted with object interaction diagrams (from the Booch method). StP also include use cases. IDE apply an adaptable meta-model so it is possible to add additional features to the languages used if this is regarded as sensible based on the current modeling situation. This can be done both in an ad-hoc fashion, and by reusing other well-known modeling approaches supported in StP. The repository of StP is implemented in a Sybase RDBMS, thus supporting standard database-functionality to ensure persistence and availability of models. An extension is that availability of models do not have to go via the database. The tool have a collaboration-window, enabling the modeler to show someone else on the network what is being done to the diagram in real-time. Most changes can be kept private until committed, but some changes need to be done directly in the repository, slightly compromising privacy. The locking strategy of StP is flexible, and enables total constraints to be applied to the workgroup activities or for constraints to be relaxed to the extent of very limited controls. The query reporting language of StP(QRL) can be used to generate reports and interrogate the repository. Repository items are time-stamped, and identify the name of the modeler. Versioning of models can be done through the link to external tools for configuration and version management such as ClearCase, SCCS, RCS, and PCMS.
- Empirical quality. It is possible to visually or logically select a set of nodes in a diagram, and move these as one. In a tree-structured diagram, the subtree moves automatically when the tree-node is moved. Connections are auto-routed, and snap-to-grid functionality is included. Scaling of symbols to be able to fit the text inside the symbol is possible. Some auto re-draw mechanisms is also included. Bridges/tunnels to show that connections cross without meeting is not supported.
- Syntactic quality: Checking scripts can be invoked to report on e.g. violation of inheritance rules and cyclic inheritance. Syntax is checked interactively. When saving files to the repository, additional error detection mechanism is invoked, centering on the problem area in the diagram. Incomplete and inaccurate models can be stored on the other hand, also at generation time. Checks for consistency between diagrams are supported, e.g. the tool provides a checking script which identifies processes, actions, and activities that are not yet represented as operations in the object

model. QRL can be used to extend these possibilities even further, e.g. to check that naming conventions are followed.

- Semantic quality: There is no formal initialization phase within the method. However, the tool provides support for requirements capture and traceability. Adding nodes to a diagram is easy. Nodes are selected from the tool bar. Once selected, an item type can be added repeatedly without re-selection. You can easily associate one item with another, by typing the name of the associated item into the appropriate field in the dictionary form. The editors can be configured to provide undo at any level. However, a commitment of data to the dictionary cannot be easily undone. StP helps you to find and reuse previously created definitions by presenting a list. As a result you are less likely to create empty definitions accidentally by getting the name wrong. The OMT-methodology suggest a set of guidelines for achieving complete and valid models. Test cases can be generated automatically from the use cases created. QRL can be used to perform translations between models at different levels (e.g. analysis to design), which involve population of the repository.
- Pragmatic quality: It is possible to create code frames and database schemas from the models, a submodel, or a class, thus supporting the creation of prototypes. Detailed code must be added manually. If you regenerate the code frames after a model change, it can preserve your handwritten code. QRL can be used to tailor the code-generation facilities. One can translate a model in one modeling language into another in a similar type of modeling language, if the audience are more familiar with this language. The tool provides help facilities for explaining the purpose and correct usage of the modeling. No explanation generation of model behavior is provided. StP supports what they call viewpoints, which is a filtering mechanism allowing you to look at a diagram from a set of separately storable perspectives. StP provides some pre-defined filters, and the modeler can also create his own. Navigation-functionality such as pan, page, infinite zoom, and the printing of a selected area and fit-to page printing is included. Items can be found by searching for certain criteria besides names. StP can also use wildcards when finding names. You can easily browse from an item to any associated item. If you navigate to an item, StP centers on that item and flashes it three times.
- Social quality: Not addressed specifically.
- Knowledge quality: Not specifically addressed in the methodology.

B. Algorithms

We present in this appendix some of the detailed algorithms used in connection with consistency checking in PPP.

B.1 Static Consistency Checking for PPM

Algorithm *Calculation-of-Input-Output-Condition*
input: *An output O of the process network;*
output: *The i/o condition expression for O on the external inputs of the network;*

begin
in the i/o conditions for the subprocesses which has output O
{ O and inputs to the subprocess are given different numbers in the sub-processes respectively}
collect all n rows for O, form the expression
$S_0 = T_1 \vee \cdots, \vee T_n$, $T_i = I_{i_1} \wedge \cdots, \wedge I_{i_j}$
{($i \geq 1$ and I_{i_j} is an input for a subprocess }
and let S_0 be the current expression;
repeat
if *in the i/o condition of a sub-process which has n_1 inputs and outputs O there is no row for O*
then begin *form the expression $S_1 = I_1 \vee \cdots \vee I_{n_1}$;*
{I_j is the name of an input to the sub-process, $1 \leq j \leq n_1$}
{just assume there are n_1 rows for O, each of which shows an input is used}
let S_{O_1} be the disjunction of the current expression and S_1;
let S_{O_1} be the current expression;
until *all such sub-processes have been found;*
repeat *do on the current expression S*
begin
if *any input I is linked to the output(s) $O_1, \cdots, O_m (m \geq 1)$ of the subprocesses in the process network through some flow(s)*
then *substitute I with $C_1 \vee \cdots \vee C_m$*
{ $Ck (1 \leq k \leq m)$ is the condition expression for O_k}
and let the new expression after the substitution be the current expression;
if *I is linked with a data store and the data store receives data from outputs $O_1, \cdots, O_v$*
then *substitute I with $C_1 \vee \cdots \vee C_v$*
{ $Ck (1 \leq k \leq v)$ is the condition expression for O_k}
and let the new expression after the substitution be the current expression;
if *I is linked with a data store and the data store does not receive any data within the process network*

then *substitute* I *with* $I_1 \vee \cdots \vee I_g$
{ $I_1, \cdots I_g$ *are all the external inputs to the process network*}
and let the new expression after the substitution be the current expression;
transform the current expression into a disjunction normal form S_n*;*
delete from S_n *all the conjunction terms whose inputs are external inputs to the network but they can not appear together in any CIP of the higher level process, and let the left parts of* S_n *be the current expression*
end
until *the content of* S *can not be changed by the above substitutions or* S *has been the same with one of the old expressions:*
delete all the conjunction terms that contain an internal input from the current expression;
let the left part of the current expression be the condition expression of O *for the process network;*
end *(algorithm)*

B.2 Constructivity Checking in PPM

Now we introduce the data structure of the state vector in a PASCAL-like syntax:

```
type state_vector =
 record
  normal_system_state : boolean;
  states_of_processes: array[1:N] of state_of_process;
  {there are N processes in the network}
  states_of_flow_pipes: array[1:M] of state_of_flow_pipe;
  {there are M flows in the network}
 end;

type state_of_process =
 record
  running: boolean;
  CIP: CIP_state; {one CIP is used during an execution of the process }
  COP: COP_state;{one COP is used during an execution of the process }
 end;

type state_of_flow_pipe =
 record
  volume: integer; {the maximum capacity of the flow}
  has_data: integer; {the sum of data staying in the flow at a moment}
 end;

type CIP_state =
 record
  ID: integer;{the identifier of the CIP in the process}
  receive_events: array[1:NI] of event; {NI possible kinds of events at the CIP}
 end;

type COP_state =
 record
  ID: integer;{the identifier of the COP in the process}
```

```
    send_events: array[1:NO] of event; {NO possible kinds of events at the COP}
   end;

  type event =
   record
    event_name: string;
    happening_times: integer; {the times that the event has happened during
    the current execution of the process }
   end;
```

Following is the algorithms for the state-transition. The first one is to check out all possible event sets at a specific state.

```
  type set_of_event_sets = set of events;
  type events = set of string; {a set of event names}
```

Algorithm *Find-Possible-Event-Sets*(S : *state_vector*, *var* $PESS$: *set_of_event_sets*))
parameter: *input: a state-vector S; output: the set of all possible event sets at the state;*

```
 begin
   PESS := ∅; {it is set to empty initially }
   {search all possible event sets}
    for each process P in the network do
    begin
     if for the process state of P in S
     S.states_of_processes[i].running = false
     { i is the index of the process state for P in S }
     then {P is idle}
     begin
      if for every triggering input of a CIP of P
      S.states_of_flow_pipes[j_v].has_data ≠ 0
      { j_v is the index of the state of the flow linked to
      the member of the triggering input group in S}
      then insert {receive_i_{j_1}, ···, receive_i_{j_l}} into PESS;
      {that P is triggered and then receives the triggering inputs is a
      possible event}
     end
     else { P is running}
     begin
      {check if P can terminate}
      if for any non-triggering and non-conditional input in the CIP and any
      non-terminate and non-conditional output in the COP,
      S.states_of_processes[i].CIP.receive_events[k_1].happening_times = e_1
      {e_1 = 1 for a singular input, ≥ 2 for a repeating input},
      { k_1 is the index of the input in S.states_of_processes[i] }
      S.states_of_processes[i].COP.send_events[k_2].happening_times = e_2
      {e_2 = 1 for a singular output, ≥ 2 for a repeating output},
      {k_2 is the index of the output in S.states_of_processes[i] }
      and e_1, e_2 satisfies the condition specified in the assumptions,
      then {P can terminate}
        insert {send_o_{m_1}, ···, send_o_{m_l}} into PESS;
        {o_{m_1}, ···, o_{m_l} are all the terminating outputs of the COP of P}
      else {P can not terminate }
```

begin
if *for a flow f which is linked with an input I in the CIP*
$S.states\_of\_flow\_pipes[h].has\_data \neq 0$ {*the linked flow has data*}
{h *is the index for the state of the flow is* S}
and $S.states\_of\_processes[i].CIP.receive\_events[c].happening\_times < e_1$
{*there is at least one input to be received* }
then
find out all such inputs $i_{l_1}, \cdots, i_{l_m}$ *and insert*
$\{receive\_i_{l_1}, \cdots, receive\_i_{l_m}\}$ *into* $PESS$
else {*no input to be received*}
begin
if *for a non-terminating output O in the COP*
$S.states\_of\_processes[i].COP.send\_events[c].happening\_times = e_2$,
{c *is the index for the output* O *in the state* S}
, e_2 *does not satisfy the condition specified in the assumptions*
and the output is ready according to the i/o condition of P
and the state of CIP of P
{*there is at least one non-terminating output to be sent out*}
then *find out all such outputs* $o_{g_1}, \cdots, o_{g_n}$ *and insert*
$\{send\_o_{g_1}, \cdots, send\_o_{g_n}\}$ *into* $PESS$
end
end
end
end { *all possible event sets for process* P *have been found*}
{*all the possible events sets for the whole process network have been found*}
end *(the algorithm)*

The second algorithm builds new state nodes from a state node S and calculate all the new states resulting from the corresponding event groups. Every new state is checked to see if it is a consistent state. If it is, then the algorithm is called recursively to calculate more possible states; otherwise it is marked as an inconsistent state.

Algorithm $State\text{-}Transition(S : state\_vector, PESS : set\_of\_event\_sets)$
parameter: *a state-vector S and the possible event sets $PESS$ at the state;*
result: *extension of the state transition diagram with possibly more nodes from the state S;*

begin
var NEW_PESS: *set_of_event_sets;*
if $PESS = \emptyset$ {*No events may happen at state* S }
then *mark the node for S as a STOP node*
else {*there are possible event sets in* $PESS$}
begin
for *each event set ES in $PESS$* **do**
begin
case $ES = \{receive\_i_{j_1}, \cdots, receive\_ij_l\}$
and the l inputs are triggering inputs to the process P:
begin
calculate all possible m CIP/COP pairs on the triggering condition
{ *a triggering group inputs may trigger more than one CIPs, and*
each CIP may produce outputs for more than one COP }
create m new nodes with state_vectors $S_{n_1}, \cdots, S_{n_m}$ *and with the*
CIP/COP pairs respectively and m edges from the node for S to them,

all marked with the
value of ES $receive\_i_{j_1}, \cdots, receive\_ij_l$;
assign value to the state_vectors to express that:
process P is now running;
the flows for the triggering inputs are now empty;
{ *the has_data items in* S_{n_k} *for the flows is 0 now* }
The events $receive\_i_{j_1}, \cdots, receive\_ij_l$ *have happened once;*
All other events for P have not happened yet;
end
case $ES = \{send\_o_{j_1}, \cdots, send\_o_{j_s}\}$
and the s outputs are terminating outputs to the process P:
begin
create a new node with state_vector S_n *and an edge from the node for S to it;*
mark the edge with the value of ES;
assign value to the state_vector S_n*:*
first copy the value of S to S_n;
then change the data so that P is idle and all other data for P is cleaned and each of the flows linked to the outputs has got a new item ;
{ *the has_data items for in S is added with 1 if the flow is not linked with a data store*}
end
case $ES = \{receive\_i_{j_1}, \cdots, receive\_ij_l\}$
and the l inputs are non-triggering inputs to the process P:
begin
create a new node with state_vector S_n *and an edge from the node for S to it;*
mark the edge with the value of ES;
assign value to the state_vector S_n*:*
first copy the value of S to S_n;
then change the data so that the has_data items and happening_times in S for the inputs and the corresponding flows changed;
{*for any receive operation, if the input is not an external and repeating input and the flow is not linked with a data store, then the has_data of the flow is reduced with 1; The happening_times for the receive event is added with 1* }
end
case $ES = \{send\_o_{j_1}, \cdots, send\_o_{j_s}\}$
and the s outputs are non-terminating outputs to the process P:
begin
create a new node with state_vector S_n *and an edge from the node for S to it;*
mark the edge with the value of ES;
assign value to the state_vector S_n*:*
first copy the value of S to S_n;
then change the data so that the has_data items and happening_times in S for the inputs and the corresponding flows changed
{*for any send operation the flow has one more ite[m if it is not linked with a data store, and the send event is added with 1* }
end;
end; {*all the new nodes have been created*}
{*now check the consistency of every new node* }
for *each new node with its state_vector* S_n **do**
if S_n *falls into one of following cases:*

```
        two CIP with different triggering inputs of an idle process may be triggered;
        a CIP of an executing process may be triggered;
        a flow contains the data more than its volume;
      then S_n.normal_system_state := false; {mark S_n as "inconsistent state"}
    {now check any possible loop structure in the STD }
    for each new consistent node with its state_vector S_n do
      if S_n is same with a state_vector S_o of a node which has exited in the STD
        being constructed, or it is same with S_o in all other part but that for
        a process P and all the flows linked with the inputs of P, and
        the events are triggering P at a different CIP with that of S_o but
        all other consequent events for P will be same with that in S_o
      then begin { a loop may have been found}
        erase the node and the edge from the node for S to it;
        attach S to the node for S_o;
        create an edge from the node of S to that of S_o, and marked the edge
        with the event set used for marking the removed edge;
      end
    { recursively call the algorithm to expand the STD }
    for each new consistent node with state_vector S_n do
    begin
      call Find_Possible_Event_Sets(S_n, NEW_PESS);
      call State-Transition(S_n, NEW_PESS);
    end
  end;
end (algorithm)
```

On the basis of the two algorithms, we now illustrate the algorithm to construct the STD of a process network which is a decomposition of a higher level process P. We will first build a $START$ node representing the initial state S when the network is idle, then collect all the triggering event for P as the possible event sets, and call the previous algorithm to construct the STD from S.

Algorithm *Construct-STD*
input: *The information for a process P and a process network N as the decomposition of P, including the canonical port structures and i/o conditions for P and all the processes in N;*
output:: *An STD which shows the possible execution cases of N;*

```
begin
  var PESS: set_of_event_sets;
  create a START node with an initial state S at which all processes in N are
  idle and all flow are empty;
  S.normal_system_state := true; {mark the state as "consistent state"}
  create n new nodes and n edges from the START node to them;
  { The higher level process P has n CIPs }
  mark any edge with the event "arrival of data at the inputs" on a particular
  CIP { the special events only be recorded at the initial state}
  for each new node do
  begin
    create a new state_vector S_n and copy the value of S to S_n;
    update the value of S_n so that the flows for a CIP of P is full;
    {every corresponding flow has a item}
    if there is any conditional input in the CIP
```

then begin
create a new node and an edge from the START node to it with the same events;
copy the state_vector S_n *to* S_{n_1} *as the state_vector of the new node;*
update S_{n_1} *so that all the flows linked with the conditional inputs are empty;*
end;
{*create another state_vector where the conditional inputs are empty*}
end *(for)*
for *each new node with a state_vector* S_{new} **do**
begin
call *Find_Possible_Event_Sets*($S_{new}, PESS$);
call *State-Transition*($S_{new}, PESS$);
end *(for)*
end *(algorithm)*

Algorithm Construct-Input-Port-For-Process-Network
input: *An STD for a process network and all other data for the network;*
output: *An input port structure for the process network;*

begin
identify all n possible paths;
for *each path* $Path_i$ $(1 \leq i \leq n)$ **do**
create a list which includes all the receive events for the inputs from external flows along the path;
from all the list $List_1, \cdots, List_n$, *select out* $List_{c_1}, \cdots, List_{c_m}$ *that the event set in each of them is not proper subset of the event set of any other list among* $List_1, \cdots, List_n$ *and any lists with exactly same members are merged into a single list;*
for *each list among* $List_{c_1}, \cdots, List_{c_m}$ **do**
create an **and** *port* P_i $(1 \leq i \leq m)$ *whose members are the inputs appear in the list;*
create an **xor** *port* $PI = xor(P_1, \cdots, P_m)$;
for *each input I from external flow* **do**
begin
if *the event for receiving I does not appear in a list and the event set of the list is a subset of the event set of any list among* $List_{c_1}, \cdots, List_{c_m}$
then *mark I as* **conditional input**;
if *the event for receiving I appears in any list and it appears twice or more in a list among* $List_{c_1}, \cdots, List_{c_n}$ *and at least it appears once outside any loop in the paths*
then *mark I as a* **repeating input**;
if *I only appears in loops*
then *mark I as a* **conditional and repeating input**;
if *I does not meet any condition above*
then *mark I as a* **singular input**;
end;
end *(the algorithm)*

C. Mathematical Symbols

In this appendix, we list the main mathematical notation used in the book.

Symbol	Meaning
$\mathcal{S}$	Set
$2^{\mathcal{S}}$	Powerset
$\#\mathcal{S}$	Cardinality i.e. number of members of a set
$\emptyset$	A set with no members
$\subset$	Proper subset of set
$\subseteq$	Subset of set
$\not\subset$	Not subset of set
$\in$	Element of set
$\notin$	Not element of set
$\equiv$	Equivalent to
$\not\equiv$	Not equivalent to
$\setminus$	Complement set
$\cup$	Set union
$\cap$	Set intersection
$\neg$	Negation
$\wedge$	Logical and
$\vee$	Logical or
$\rightarrow$	Implication
$\Diamond$	Sometime in past
$\Diamond$	Sometime in future
$\blacksquare$	Always in past
$\Box$	Always in future
$\bullet$	Just before
$\circ$	Just after
$\mathcal{U}$	Until
$\mathcal{S}$	Since
τ	Trigger
ϕ	Condition

ϕ_s	State condition
ψ	Consequence
ψ_a	Action
ψ_s	State
ρ	Role
α	Actor
$\mathcal{E}$	$\neg \bullet \tau \wedge \tau \wedge \phi$
∇	Deontic operator
O	Obligatory
R	Recommended
P	Permitted
D	Discouraged
F	Forbidden
$\nabla_\rho\ (\psi\ /\ \neg \bullet \tau \wedge \tau \wedge \phi)$	General rule
N	Necessary
E	Excludes
t_R	Real time
t_M	Temporal module time
$\mathcal{A}$	Audience, the technical and social actors that must relate to a model
A_i	A member of the audience
$\mathcal{D}$	The set of all statements that can be stated about a problem at hand
$\mathcal{I}$	The set of all statements which the audience think that a model consist of
$\mathcal{K}$	The set of statements regarded relevant among the participants.
$\mathcal{K}_i$	All possible statements that would be correct and relevant for addressing the problem at hand according to the explicit knowledge of the participant A_i
$\mathcal{K}^i$	The statements of the explicit internal reality of the social actor A_i
$\mathcal{L}$	The statements that can be expressed in a given (set of) languages
L_i	A language
$\mathcal{L}_F$	Statements expressible in a set of formal (operational) languages
$\mathcal{L}_I$	Statements expressible in a set of informal languages
$\mathcal{L}_S$	Statements expressible in a set of formal (logical) languages
$\mathcal{L}_S$	Statements expressible in a set of semi-formal languages
$\mathcal{L}_i$	The statements that can be stated in language L_i
$\mathcal{M}$	The set of statements in an externalized model
$\mathcal{M}_E$	The set of explicit statements in an externalized model
$\mathcal{M}_I$	The set of implicit statements in an externalized model
$\mathcal{M}_i$	A model based on the knowledge of social actor A_i
$\mathcal{M}_{L_i}$.	A model written in language L_i
$\mathcal{M}^i$	The statements in a model which are relevant for audience member A_i
$\mathcal{M}(L_i)$	The language model of language L_i
$\mathcal{M}(\mathcal{D})$	The model of a given domain
$\mathcal{P}$	The individual social actors of the audience

$\mathcal{S}$	When used about actors, the stakeholders to the modeling, else a general set
$\mathcal{V}$	A viewspec of a model

D. Terminology

We give in this appendix a comprehensive overview of the terminology used in the book. We have also included some of the abbreviations being used.

Terms will be written in *italic* type when first defined, and will also be written in *italic* when they are used as part of other definitions. The terminology is in particular based on [235, 350, 388]. It is also influenced by our philosophical stance of social construction. This appears especially in the definitions of the basic terms, although the definitions themselves are written in a categorical style for us to be able to use the terms consistently. A constructivistic view is also followed in FRISCO, a group within IFIP WG 8.1. on design and evaluation of information systems which are trying to establish a framework for information systems terminology [126, 235]. Similarly to FRISCO, we have used a set-theoretic approach, although this is not emphasized here.

Terms are grouped in the following areas:

- Time.
- Phenomena.
- State and rules.
- Data, information, and knowledge.
- Language and models.
- Actors and activities.
- Systems.
- Social construction.
- Methodology.

In some cases the definition of a term is found after that it has been used in another definition. An alphabetical overview of the terms are given separately at the end of the appendix.

D.1 Time

The definitions in this area are to a large extent pragmatic, to be able to use them in a well-defined manner in the later definitions. Thus, we are not entering into philosophical and quantum mechanical aspects of the nature of time.

Time points . Time can be represented by time points, such that the only relation between time points other than identity is that one time point precedes the other.

Time interval . A time interval is the ordered pair of *time points* (the begin and end-point of the interval) such that the first either precedes or is equal to the other.

Time scale . A time scale divides *time* into coherent *time intervals*.

Time unit . The smallest *time interval* that can be represented on a given *time scale* is termed the time unit.

Duration . The duration of a *time interval* is the number of consecutive *time units* between the one after the one in which the begin point of the *time interval* occurs, until and including the *time unit* in which the end point of the *time interval* occurs. A *time point* has no duration.

Figure D.1 illustrates the terms discussed in this section.

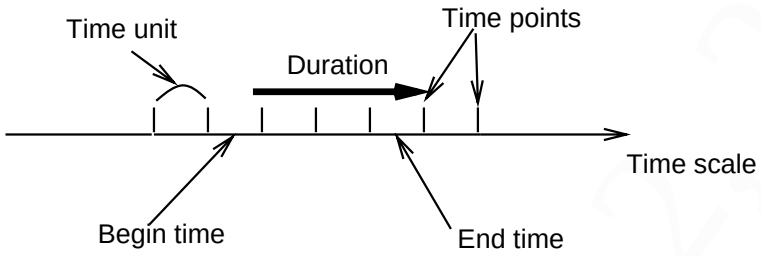


Fig. D.1. Time and duration

D.2 Phenomena

A phenomenon is used as the elementary unit of the terminology. In other similar terminologies, the term 'thing' is used in this respect [235, 388].

Phenomenon . A phenomenon is something as it appears in the mind of a person. The world is perceived by persons to consist of phenomena. A phenomenon can be perceived to exist independently of the perceiving person, or be perceived to be a purely mental.

With the phrase 'something which is perceived to exist independent of the perceiver', we mean something that the person in question regards as existing external to him, for instance another person. An idea of a new CIS that only appears in someone's mind though have no such "real-world" equivalence, at least not until it is externalized.

Relevance . A *phenomenon* is of relevance to a non-empty set of persons in a *time interval* if it is of interest to all members of the set in the *time interval*.

Potential relevance . A *phenomenon* is of potential relevance to a non-empty set of persons in a *time interval* if it is of interest to at least one of the persons in the set in the *time interval*.

Relevance is socially and temporally constrained which are as expected taking social construction into account. Relevance needs the notion of shared explicit knowledge (see below) to be meaningful i.e. if there are no phenomena which are perceived equally by two persons, no phenomena will be relevant.

Domain . A domain is defined as the source of any kind of mapping.

'Domain' includes the meaning known from algebra, but the term 'mapping' is used in a slightly more general sense than usual. Not only sets can be mapped into sets as in mathematics, but also areas into areas. When used in the mathematical sense, a domain will be a finite or infinite set of values.

Property . A property is an aspect of a *phenomenon* which can be described and given a value. A *phenomenon* will have a set of *potentially relevant properties*. The values for the properties are members of the *domains* for these properties. All *phenomenon* have at least one property, namely its perceived individual existence or lack thereof.

Type . A non-empty set of *properties* which together characterize certain *phenomena*.

Subtype . A subtype S of a *type* T is a set of *properties* such that T is a proper subset of S.

Supertype . A subset of a type

Class . The set of all *phenomena* of a certain *type*. These *phenomena* are called the *members* of the class.

Subclass . The subclass S of a class T is the proper subset of the class T such that the phenomenon in S has a type which are a subtype of the type of the phenomena of T.

When having several sub-classes of a class you can have different cases based on coverage and disjointness.

A set of *subclasses* of a *class cover* the *class* if all *members* of the *class* are *members* of at least one of the *subclasses*.

A set of *subclasses* of a *class* are *disjoint* if no *members* of a *subclass* are *members* of any of the other *subclasses* of the *class*.

A set of *subclasses* which are both *disjoint* and *cover* the *class* is called a *partition* of the *class*.

Environment . The environment of a *phenomena* is the set of *actors* which *acts upon* it.

D.3 State and Rules

State . The state of a *phenomenon* is the set of mappings of all *properties* of the *phenomenon* into values from the *domain* of the *properties*. A *phenomenon* can only be in one state within a *time unit*.

State space . The state space of a *phenomenon* is the set of all possible *states* of the *phenomenon*. All *subsystems* of a *system* have its own state space.

Transition . A transition is a mapping from a domain comprising *states* to a co-domain comprising *states*.

Event . An event is a change of *state* of a *phenomena*. It is effected through a *transition*. An event happen within a *time unit*, i.e. it has a zero *duration*.

Trigger . A trigger is a relationship between an *event* and one or more *activities* and expresses the perceived cause for an *actor* to carry out the *activities*.

History . The history of a *phenomenon* is the chronologically ordered *states* of the *phenomenon*.

Rule . A rule is something which influences the *actions* of a non-empty set of *actors*. A rule is either a rule of necessity or a deontic rule [393].

The term rule is used to cover more situations than what is usually found, since it also includes what is often referred to as goals, guidelines or instrumental rules [294].

Rule of necessity . A rule of necessity is a *rule* that must always be satisfied. It is either analytic or empirical (see below).

Analytic rule . A *rule of necessity* which can not be broken by an intersubjectively agreed definition of the terms used in the rule is called analytic.

Example: 'The age of a person is never below 0'

Empirical rule . A *rule of necessity* that can not be broken according to present *shared explicit knowledge* is called empirical.

Although not as strongly necessary as an analytic rule, this kind of rules are rules that can be treated as if they are rules of necessity, and one would not expect them to be broken.

Example: 'Nothing can travel faster than the speed of light'

Deontic rule . A *rule* which is only socially agreed among a set of persons. A deontic rule can thus be violated without redefining the terms in the rule. A deontic rule can be classified as being an obligation, a recommendation, a permission, a discouragement, or a prohibition [214].

'$\delta\epsilon o\upsilon\tau\omega\varsigma$' is Greek and means " as it should be". The inclusion of recommended and discouraged above is novel compared to traditional deontic logic [385], but has been included in newer frameworks for deontic logic e.g.[191, 192].

Constitutive rule . A *deontic rule* which applies to *phenomena* that exist only because the rule exist [327].

Generally, this kind of rule can be written: A counts as B in context C. When using a general rule-format, the context is included in the precondition.

Static rule . A *rule* restricting the allowable *states* of a *phenomenon* is called static.

Dynamic rule . A *rule* restricting the allowable *state transitions* of a *phenomenon* is called dynamic.

Temporal rule . A *rule* referring to the situation of more than one *state*

Both rules of necessity and deontic rules can be classified as being static or dynamic.

Lawful transition . A *transition* is lawful if it obey the *dynamic rules of necessity* regarding the *phenomenon*.

Deontic transition . A transition is deontic if it is *lawful* and also obey the *dynamic deontic rules* regarding the *phenomenon*.

Lawful state space . The set of *states* of a *phenomenon* that comply with the *static rules of necessity* concerning the *phenomenon* is termed the lawful state space.

Deontic state space . The set of *states* of a *phenomenon* that are *lawful* and in addition comply with the *static deontic rules* concerning the *phenomenon* is termed the deontic state space.

Internal event . An *event* that arises in a *phenomenon* by virtue of a *lawful or deontic transition* in the *phenomenon* is called internal.

External event . An external event is an *event* that arises in a *phenomenon* by virtue of the *act* of an *actor* in the *environment* of the *phenomenon*.

Stable state . A *state* in which a *phenomenon* will remain unless forced to change by virtue of an *external event*.

Unstable state . A *state* that will change into another *state* by virtue of an *internal event* is called unstable.

D.4 Data, Information, and Knowledge

Knowledge . Knowledge is the justified true belief of a person.

Knowledge is by definition linked to the individual person. It can be divided into *explicit* and *tacit knowledge*.

Explicit knowledge . Explicit knowledge is the awareness of a person of *properties* and values of *properties* of *phenomena*.

This indicates that it is both explicit knowledge to be aware of a persons height in cm, and that 'height' is a *relevant property* of a person. Since also a person is a *phenomenon*, the explicit knowledge he or she has can be looked upon as part of the *state* of this *phenomenon* since a *potential relevant property* of a person can be that he is able to know something, and what he or she knows. Through infinite introspection this could indicate that the knowledge of a person is infinite, but since it is seldom relevant with this kind of introspection, this is not found problematic. Explicit knowledge can be more or less *precise*, *certain*, and *complete*.

Example on precision: That someone knows that a city has 2,433,775 inhabitants at a certain time is more precise explicit knowledge than if the same person know that a city has around 2.5 million inhabitants.

Example on completeness: That someone knows that Oslo is the capital of Norway is less complete knowledge of Oslo than to also know that Oslo is located in the southern part of Norway.

Example on certainty: A poor farmer in Kuala Lipis who once have heard about the city of Bombay and that it lies in India has less certain explicit knowledge than someone who have been there himself. If you hear something several times from several different people, your certainty of some explicit knowledge will usually increase. If you have only read it once in a tabloid newspaper, your certainty of the "fact" is usually lower.

In FRISCO [235], explicit knowledge is termed information. We define the term 'information' differently below.

Shared explicit knowledge . Shared explicit knowledge is an inter-subjectively agreed identical awareness of some *properties* and values of *properties* of *phenomena* by two or more persons which have been achieved through a process of social construction.

Tacit knowledge . Tacit knowledge is *knowledge* that can not be represented externally to the person and only shows up in the *actions* of the person having the knowledge.

It is possible to differentiate between two kinds of tacit knowledge. That which could have been represented externally, but which one either choose not to, or can not find the appropriate *symbols* for, and so-called true tacit knowledge.

Information . Information is externalized *explicit knowledge* which are not already known by the person who receives it, i.e. a *state transition* for a person appears when he receives information, thus receiving information can be looked upon as an *event*.

This means that information is socially and temporally constrained. If I already know something (and know that I know it) , I do not perceive to receive information if I am told the same thing again (even if the certainty of

the knowledge might increase). Thus our definition is hopefully close to the one used in everyday language as illustrated in [235] "Information is what you get or may get if you ask certain kinds of questions.... Answers to such questions are often provided at some information desk."

Symbol . The *explicit knowledge* of a person can be externalized in a persistent form using symbols.

Message . A set of related *symbols* expressed in a *language* transmitted by an *actor* intended for a non-empty set of *actands*. The set of *actands* which ultimately receives the message can be empty.

Communication . The exchange of *messages* between *actors*.

Sign . A sign is the triplet (*symbol*, person, *phenomenon*), i.e. a sign is *symbol* that represent a *phenomenon* for a person.

Data . Data are *symbols* that can be preserved, transformed, and transported by a computer. Data and other *symbols* can be internalized as *knowledge* by persons.

D.5 Language and Models

Language . A set of *symbols*, the *graphemes* of the language being the smallest units in the writing system capable of causing a contrast in meaning, a set of *words* being a set of related *symbols* constituting the *vocabulary* of the language, rules to form *sentences* being a set of related words (*syntax*), and some inter-subjectively agreed definitions of what the different sentences mean (*semantics*).

In a natural language e.g. English, the symbols and words will be ordered linearly, whereas in a two-dimensional language symbols are ordered spatially. In addition to the aspects described above, one also often talks about the *pragmatics* when discussing languages, being the relationship between symbols, words, and sentences and the effect these have on persons.

Statement . A *sentence* representing a single *property* of a certain *phenomenon*.

Language extension . The set of all *statements* that can be made according to the *graphemes*, *vocabulary*, and *syntax* of a *language*.

Natural language . A natural language is the *language* of a cultural society (for instance a tribe or a nation) - It is usually learned and applied from childhood by the set of persons belonging to the society.

Professional language . A professional language is a *language* used by a set of persons working in a certain kind of area or in a scientific discipline. Usually such a language is not learned before the person has been active in the area for a while.

Formalism . A formalism is a formal *language*, i.e. a *language* with a precisely defined *vocabulary*, *syntax*, and *semantics*.

The semantics can be *operational* and/or *logical*. If the semantics is based on mathematical logic, we use the term *logical formalism*. If it is possible to execute a set of sentences in the language on a computer, the language is said to have an operational semantics.

All formalisms are professional languages.

Semi-formal language . A semi-formal language is a *language* with a precisely defined *vocabulary* and *syntax*, but without a precisely defined *semantics*.

Also semi-formal languages (e.g. DFD) are professional languages.

Informal language . An informal language is neither *formal* nor *semi-formal*. *Natural languages* are of this category, and also a *professional language* can be informal.

Abstraction . An abstraction is the *phenomenon* of a set of *phenomena* and its *properties* at some level of approximation. The abstraction contains incomplete *explicit knowledge* about the *phenomena*, i.e. there are more to know about the *phenomena* than is in the abstraction. This do not mean that the abstraction can not contain all *relevant knowledge* in a given *time interval*.

Classification . The *abstraction* where individual *phenomena* are grouped together in a *class* based on perceived common *properties*.

Example: "Rod Steward" and "Mick Jagger" can be grouped together in the class "singers".

Aggregation . An *abstraction* which is a Cartesian product of *classes*.

Example: A bicycle being built up from wheels, a seat, a frame, handlebars etc.

Generalization . An *abstraction* which is a subset of the union of a set of *classes*.

Example: Both employees and customers are persons.

Association . An *abstraction* which is a set of *classes*.

Example: The classes "Men" and "Woman" are members of the set "sex-groups".

Model . A model is an *abstraction* externalized in a *language*.

A model is assumed to be simpler than, resemble, have the same structure and way of functioning as the phenomena it represent.

Conceptual model . A *model* of a *domain* made in a formal or semi-formal language with a limited vocabulary.

Comment: Many conceptual modeling languages are partly diagrammatic, in which case they are are combination of logographic and iconographic, but this is not looked upon as a requirement. Some conceptual modeling languages also have aspects that are pictographic.

Language model . The *model* of a *language*.

Within conceptual modeling, this is often termed 'meta-model', which is only a proper term when looking upon it from the point of view of repository-management for a modeling-tool where the instantiation of the model is another model in the same or a different modeling language.

System model . A *model* of a *system*.

D.6 Actors and Activities

A *phenomenon* is *acted* upon by another *phenomenon* if its *history* is different from what it would have been if the other *phenomenon* did not influence it.

Actor . An actor is a *phenomenon* that *acts upon* another *phenomenon*, the *actand* .

Acquaintances . The acquaintances of an *actor* is the set of *actors* that either *acts upon* or is *acted upon* by the *actor*.

Social actor . A social actor is an *actor* that includes at least one person. Social actors might be *individual* or *organizational* (see below).

Technical actor . A technical actor is an *actor* that do not include any persons.

Technical actors can be *computational* and *temporal* . Other subtypes of actors might for instance be production actors, but these will not be discussed here.

Whereas temporal actors are some time-measuring device (i.e. a clock of some sort), computational actors are either *hardware* actors or *software* actors. *Computational actors* are either *atomic* or *systemic* including *atomic* and *systemic subsystems*. Computational actors can be said to be compatible in the following meanings:

- *Hardware compatibility:* Stating which *hardware actors* that can *act* upon each other.
- *Executional compatibility:* Describe which *software actors* that can be executed on which *hardware actors*.
- *Software compatibility:* Stating which *software actors* that can *act* upon each other.

Software actors can be versions of 0:N other software actors, i.e. a software actor can be recreated by performing a set of state changes to the actor it is a version of. A set of state changes in this meaning is called a *delta*. The original actor is called a *predecessor* of the version actor, whereas this is called a *successor* of the original actor. Software actors might have several predecessors and successors. These relations are transitive. Two or more software actors that have the same immediate predecessor are termed *variants* .

Software actors are *supportive* or **applicative** relative to an *organization*. The difference is that applicative actors are being customized to some degree to cater for the specific needs of the organization. For instance will a customized order-entry application system be regarded as applicative, whereas an underlying commercial database system which is used by the order-entry system is supportive. Subtypes of supportive actors are [302]:

- *General supportive software actors* are *software actors* that potentially support all *social actors* in an *organization*. Examples of classes of these are operating systems, file-handlers, and window-systems.
- *Office supportive software actors* are *software actors* supporting office work, example of classes of these are word processors, database systems, graphics tools, spreadsheets, communication programs, and statistics programs.
- *Developer supportive software actors* are *software actors* that typically only support developers of *applicative software actors* directly. Examples of classes of such systems are compliers, CASE-tools, debuggers, general modeling tools, building tools, versioning tools, and test tools.

Internal actor . *Actors* being *internal* to an *organization* are *actors* being part of the *organizational system* of the *organization* in one or more of the relevant *roles* they are currently filling.

External actor . *Actors* being *external* to an *organization* are *actors* not being part of the *organizational system* of the *organization* in any of the relevant *roles* they are currently filling.

Individual social actor . A person interacting with his *environment* is termed an individual social actor.

Organizational social actor . An organizational actor is a *social actor* which consists of a set of more than one person performing goal-oriented and coordinated action. An organizational actor can also include *technical actors*, but this is not mandatory.

Permanent organizational actor . An *organizational actor* for which a *begin time-point* of its existence can be perceived, but normally not the future *end time-point*.

Temporary organizational actor . An *organizational actor* for which both the *begin time-point* and the possibly future *end time-point* of its existence can be perceived.

Periodic organizational actor . An *organizational actor* for which a set of *begin time-points* and (possibly future) *end time-points* of its existence can be perceived, and where there is normally the same *time-interval* between the different *begin time-points*. The *duration* of this *time-interval* is longer than the individual lifetime of the *organizational actor*.

Reincarnation . The creation of a *periodic organizational actor*.

Action . An action is the *phenomenon* of one *phenomenon acting* upon other *phenomena*.

Activity . An activity is a *system* of *actions*.

Stakeholder . The stakeholders of an *activity* are the set of persons who perceive or is perceived by other persons to potentially lose or gain from the *activity*.

Participant . The participants of an *activity* are the set of persons who act upon the *actands* of the *activity* as part of the *activity*.

Process . A process is an *activity* which takes a set of *phenomena* and transforms them into a possibly empty set of *phenomena*.

Behavior . Behavior is defined as a time series of *activities*.

Role . *Behavior* that can be expected by an *actor* by other *actors*.

Agent . An *actor* acting in a particular *role*.

Formal role . A *role* where part of the expected *behavior* of an *actor* filling the *role* is institutionalized by an *organizational actor*. A typical example of a formal role is a position such as a professor. All roles have usually also two additional aspects:

- The informal part of the *role*. Expectations to an *actor* filling the *role* which are not institutionalized. e.g. a professor is absent-minded.
- The expectation to an *agent*, because of the particular *actor* filling the *role*.

Role conflict . Inconsistent expectations to an *actor* because of filling two or more *roles* or because of differing expectations to a role that the actor fill from two or more other actors.

D.7 Systems

System . A system is a set of correlated *phenomena*, which itself is a *phenomenon*. Each *phenomenon* that is contained in the system is said to be *part* of the system. A system has at least one systemic *property* not possessed by any of its parts.

The following example taken from [235] indicates the necessity of the requirement of a systemic property: If you buy some eggs from a farmer and use two of them for breakfast, then the correlated phenomena: You, the farmer, the farmers hen that laid the eggs, the frying pan you used to prepare the eggs, and the two eggs now in your stomach could fit as a system using a definition not including a systemic property.

System viewer . A person who perceives the *system* as a *phenomenon*.

Subsystem . A subsystem of a *system* is a *system* that is part of another *system*, the set of *phenomena* being part of the subsystem is a proper subset of the set of *phenomena* being part of the whole *system*.

Subsystem structure . A *partition* of a *system* into a set of *subsystems* together with a set of correlations among the *subsystems*.

Constructivity . The *relevant properties* of a *system* can be derived from the *relevant properties* of the *subsystems* of the *system* given the *subsystem structure* of the *system* [350].

Constructivity should not be confused with social construction theory.

Active system . A *system* where at least one of the *subsystems* is an *activity* is called an active system.

Passive system . If there is no *subsystem* in the *system* that is an activity, it is called a passive system.

Open system . A *system* is open if it has an *environment*.

Information system (IS) . An information system is a *system* for the dissemination of *data* between persons, i.e. to potentially increase their *knowledge*.

Data system . A data system is a *system* to preserve, transform, and transport *data*.

A data system is usually a sub-system of an *information system*. Both data systems and *information systems* may be contained in the *domain* they convey *data* about.

Organization . An organization is defined as a non-empty set of persons, and other *phenomena* which is a *phenomenon* where goal-oriented and coordinated *action* is aimed at.

An organization is an *organizational actor* when interacting with other *phenomena*.

Organizational system . An organizational system is a *system* having the *actors* and *activities* of an *organization* as *subsystems*.

Organizational information system (OIS) . An OIS is the *information system* for the dissemination of *data* within an *organization*. The OIS is a *subsystem* of an *organizational system*.

Computerized information system(CIS) . A CIS is an *information system* which are based on the use of computers for the dissemination of *data*.

User (of a CIS) . A user of a *CIS* is someone who potentially increases his or her *knowledge* about some *phenomena* other than the *CIS* with the support of the *CIS*. An *end-user* increases his and hers *knowledge* in areas which are *relevant* to him independently of the actual *CIS* by *interacting* with the *CIS*. *Indirect users* of a *CIS* increase their *knowledge* by getting results from the *CIS* without directly *interacting* with the actual *CIS*.

Computerized organizational information system (COIS) . A COIS is a *system* for the dissemination of *data* within an *organization* which are based on the use of computers. This is a *subsystem* of the *OIS* of the organization.

The COIS contains the set of *internal software actors* which support the *internal social actors* of the *organization*, and the *hardware actors* these *software actors* are executed on.

Application system . An application system is a *subsystem* of the *COIS* being adapted to the needs of the *organization*.

When an application system interact with its *environment* it is an *applicative actor*.

(Application system) portfolio . The portfolio of an *organization* is the set of *application systems* in the *COIS* of the *organization*.

Dynamic system . A *system* that always is in a *state* from which there exist a *lawful transition*.

Static system . A *system* that is not *dynamic* is called *static*.

D.8 Social Construction

Definitions of terms from social construction theory as they are used in the book are given here. The definitions are based on [136].

Local reality . The *local reality* of a person is the way the person perceives the world that he or she *acts* in.

In addition to the persons explicit and tacit *knowledge* this also includes feelings and values of the person.

Externalization . The enactment of *local reality*. The most important ways *social actors* externalize their *local reality*, are to speak and to construct *languages*, artifacts such as *models*, and institutions such as *rules*.

Organizational reality . That which guides and controls persons *actions* in an *organization*.

Internalization . Making sense out of the *actions*, institutions, artifacts etc. in the *organization*, and making this *organizational reality* part of the individual *local reality* of a person.

Organizational closure . A *process* of social construction where the *actors* keep reproducing the same *organizational reality*.

D.9 Methodology

The terms underneath are defined here in the context of conceptual modeling for CIS-support in organizations.

Conceptual modeling . The *activity* of constructing *conceptual models*.

Audience . The *actors* that need to relate to the *conceptual models* constructed during *conceptual modeling*

Method . A method is a set of *rules* for creating *models* with a *language*.

Approach . An approach consists of a non-empty set of semi-formal or formal *languages* and a number of *rules* for using these *languages* to construct *models*.

(Model) verification . The *process* of assuring whether a *model*, created according to a certain *approach*, conforms to the *rules of necessity* of the *language* used, or has the expected *semantic*.

(Model) validation . The *process* of assuring that a *model* corresponds to the *explicit knowledge* of those *social actors* which are the source of the *model* .

Whereas verification is potentially decidable, validation is not so, one can never be 100% certain that the externalization in the form of a conceptual model correspond to the local reality of an individual. Even though, validation is a useful activity, due to the possibility for falsification, i.e. one can say that a model do not correspond to one's internal reality.

(Model) transformation . A *process* where a *model* written in a *language* is transformed into another *model* in the same *language*.

Statement insertion . A *transformation* where the resulting *model* contains *statements* that are not contained in the original *model*.

Statement deletion . A *transformation* where the resulting *model* do not contain *statements* that are contained in the original *model*.

Syntactically valid statement deletion . A *statement deletion* resulting in a *model* being conformant to the *syntax* of the *language* the *model* is written in.

(Model) layout modification . *Transforming* a *model* into another *model* containing the same *statements*.

(Model) filtering . *Transforming* a *model* into another *model* containing a subset of the *statements* of the original model.

A model filtering consist of a set of *statement deletions*.

Syntactically valid (model) filtering . *Model filtering* resulting in a *model* being conformant to the *syntax* of the *language* of the model.

(Model) translation . A *process* where a *model* written in a *language* is transformed into another *model* written in a (set of) different *language(s)*.

Rephrasing . A *transformation* where some of the implicit *statements* of a model are made explicit.

Paraphrasing . A *translation* where the involved *languages* are textual.

Visualization . A *translation* where the source *language* is textual, and the target *language* is diagrammatic.

Code-generation . A computer-supported *translation* where the target *language* have an *executional semantics* for which there exist tool for execution.

Complete translation . A *translation* where all *statements* in the source *model* is also contained in the target *model*.

Valid translation . A *translation* in which all *statements* in the target *model* is also contained in the source *model*.

Prototype . An executable *model* of (parts of) an *information system* which emphasizes specific aspects of that *system*.

Development of an application system in an organization . The *process* of producing a new *application system* in the *organization* based on the current *OIS* and the *knowledge* of *internal* and potentially *external actors*.

Development is divided into two categories.

- Development of *replacement systems* being *application systems* that replace existing *application systems*, and offer the same functionality as the already existing *application systems*.
- Development of *application systems* covering functional areas that are not currently supported by the existing *COIS*.

Maintenance of an application system in an organization . The *process* of creating an updated *version* of an *application system* used in the *organization* through a temporally ordered set of *lawful transitions* based on an existing *application system* and the knowledge of internal and potentially external *actors*.

Corrective maintenance . *Maintenance* performed to identify and correct processing failures, performance failures and implementation failures in an *application system*.

Adaptive maintenance . *Maintenance* performed to adapt *application systems* to the changes among the *supporting technical actors* of the *application system*.

Perfective maintenance . *Maintenance* performed to enhance performance, change or add new functionality, or improve future maintainability of the *application system*. Perfective maintenance is divided into *functional and non-functional perfective maintenance* based on the effects of the performed changes. Functional changes are changes to the functions offered by the *application system* or said differently, how *users* can potentially increase their *knowledge* using the *application system*. Non-functional changes implies changes where the quality features of the *application system* and other features being important for the developer and maintainer of the *application system* such as modifiability are improved.

Functional development . *Development* or *maintenance* where changes in the *application system* increases the functional coverage of the *portfolio* of the *organization*. This includes *development* of new *application systems* which covers areas which are not covered by the existing *COIS*, and also includes *functional perfective maintenance*.

Functional maintenance . Work made to sustain the functional coverage of the *portfolio* of the *organization*. This includes the three other types of *maintenance*, but also includes the *development* of *replacement systems*.

Devtenance in an organization . The *process* of producing an updated *version* of the *COIS* through a temporally ordered set of *lawful transitions* based on the existing *OIS* and the *knowledge* of *internal* and potentially *external actors*.

Methodology . A *system* of *rules*, *approaches*, and *computational actors* to aid *development and/or maintenance* of *application systems*.

D.10 Abbreviations

This section contains a set of abbreviations used in the book. These are listed here in full, although we are not giving any further explanation of the term.

4GL . 4. generation language.

ABC . Actor, bank, channel (in ABC-models).

ABC . Alle bidrar til consensus (in ABC-method).

AD . Actor dependency.

ALBERT . Agent-oriented language for building and eliciting real-time requirements.

AM . Actor model.

ARIES . Acquisition of requirements and incremental evolution of specifications.

BNF . Backus-Naur Form.

BNM . Behavioral network model.

CASE . Computer aided software engineering.

CATWOE . Customer, actor, transformation, weltanschauung, owner, environment.

CFG . Context-free phrase grammar.

CFP . Call for papers.

CIM . Computer integrated manufacturing.

CIP . Canonical input port.

CML . Conceptual modeling language.

COISIR . *COIS* investigation report.

COP . Canonical output port.

CR . Change request.

CRC . Camera ready copy.

CSCW . Computer supported cooperative work.

DAIDA . Development of advanced interactive data intensive applications.

DBMS . Data base management system.

DND . Den norske dataforening.

DRL . Deontic rule language.

DSM . Domain structure model.

ECML . Executable conceptual modeling language.

EDFD . Entity data flow diagrams.

EIS . Existing information system.

ER . Entity relationship.

ERAE . Entity, relationship, attribute, event.

ERL . External rule language.

ERT . Entity relationship time.

ESPRIT . European strategic programme for research and development in information technology.

F3 . From fuzzy to formal.

FCIS . Future CIS.

FG . Functional grammar.

FRISCO . Framework for information systems concepts.

FSM . Finite state machine.

GSM . Generic semantic model.

HCI . Human computer interface.

HOQ . House of quality.

IBIS . Issue based information systems.

ICASE . Integrated *CASE*.

IDT . Institutt for datateknikk og telematikk.

IFIP . International federation of information processing.

ISO . International standards organization.

JAD . Joint application design/development.

JSD . Jackson System Development.

NATURE . Novel approaches to theories underlying requirements engineering.

NFR . Non-functional requirement.

NTH . Norges teknisk høyskole.

OMT . Object modeling technique.

ONER . Our new *ER* modeling language.

OO . Object-oriented.

OOA . Object-oriented analysis.

OOD . Object-oriented design.

OORASS . Object-oriented role analysis, synthesis, and structuring.

PD . Participatory design.

PID . Process interaction diagram.

PLD . Process life description.

PPM . Process port modeling.

PPP . Phenomena, process, program.

RDD . Responsibility driven design.

RST . Rhetorical structure theory.

SAMPO . Speech act based office modeling approach.

SA/RT . Structured analysis, real time.

SASD . Structured analysis, Structured Design.

SCM . Software configuration management.

SD . Standard deviation.

SDL . Semantic data language.

SQL . Structured query language.

SSADM . Structured systems analysis and design method.

STD . State transition diagrams.

SSM . Soft systems methodology.

UDD . User interface dialog description.

UID . User interface description.

UIP . User interface presentation.

VDM . Vienna design method.

References

1. S. Abu-Hakima and F. Oppacher. Improving explanations in knowledge-based systems: RATIONALE. *Knowledge Acquisition*, 2(4):301–343, 1990.
2. M. Ader, G. Lu, P. Pons, J. Monguio, L. Lopez, G. De Michelis, M. A. Grasso, and G. Vlondakis. Woorks, an object-oriented workflow system for offices. Technical report, ITHACA technical report available from ftp://cui.unige.ch/OO-articles/ITHACA/WooRKS, 1994.
3. A. V. Aho, R. Sethi, and J. D. Ullman. *Compiler : Principles, Techniques and Tools*. Addison-Wesley, 1986.
4. M. Alavi. An assessment of the prototyping approach to information systems development. *Communications of the ACM*, 27(6):556–563, June 1984.
5. M. Alford. *Software Requirements Engineering Methodology (SREM) at the Age of Eleven - Requirements Driven Design*, chapter 11.
6. J. Allwood and L. G. Andersson. *Semantik (In Swedish)*. Institute of Linguistics, University of Gothenburg, 1976.
7. R. Andersen. *A Configuration Management Approach for Supporting Cooperative Information System Development*. PhD thesis, IDT, NTH, Trondheim, Norway, 1994.
8. R. Andersen, J. A. Bubenko jr., and A. Sølvberg, editors. *Proceedings of the Third International Conference on Advanced Information Systems Engineering (CAiSE'91)*, Trondheim, Norway, May 1991. Springer-Verlag.
9. Andersen Consulting. *Method/1, System Development Management*, 1995.
10. R. S. Arnold and D. A. Parker. The dimensions of healthy maintenance. In *Proceedings of the 6th International Conference on Software Engineering (ICSE)*, pages 10–17. IEEE Computer Society Press, September 13–16 1982.
11. Ascent Logic Corporation. *RDD-100 Requirements Driven Development*, 1993.
12. P. Atzeni and R. Torlone. A metamodel approach for the management of multiple models and the translation of schemas. *Information Systems*, 18(6):349–362, 1993.
13. J. H. August. *Joint Application Design: The Group Session Approach to System Design*. Yourdon Press, 1991.
14. E. Auramäki, R. Hirschheim, and K. Lyytinen. Modelling offices through discourse analysis: The SAMPO approach. *The Computer Journal*, 35(4):342–352, 1992.
15. J. L. Austin. *How to do things with words*. Harvard University Press, Cambrige MA, 1962.
16. D. E. Avison and A. T. Wood-Harper. *Multiview: An Exploration in Information Systems Development*. Blackwell, Oxford, England, 1990.
17. N. A. Baas. Hierarchical systems. Foundations of a mathematical theory and application. Technical report, Department of mathematics, The university of Trondheim, Norway, 1976.

18. S. C. Bailin. An object-oriented requirements specification method. *Communications of the ACM*, 32(5):608–623, May 1989.
19. R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, November 1985.
20. R. M Balzer, N. M. Goldman, and D. S. Wile. Operational specifications as the basis for rapid prototyping. *ACM SIGSOFT Software Engineering Notes*, 7(5):3–16, December 1982.
21. V. R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, 7(1):19–25, January 1990.
22. A. Basu and R. Ahad. Using a Relational Database to Support Explanation in a Knowledge-Based System. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):572–581, December 1992.
23. G. Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: An annotated bibliography. Technical report, Brown University, anonymous ftp to wilma.cs.brown.edu , /pub/papers/compgeo/gdbiblio.ps.Z, June 1994.
24. K. M. Benner. The ARIES simulation component (ASC). In *Proceedings of the Eight Knowledge-Based Software Engineering Conference (KBSE'93)*, September 1993.
25. E. Berg, J. Krogstie, and Ø. Sandvold. Enhancing user participation in system design using groupware tools. In *Proceedings of IRIS'20*, pages 481–500, 1997.
26. P. Berger and T. Luckmann. *The Social Construction of Reality: A Treatise in the Sociology of Knowledge*. Penguin, 1966.
27. L. Bergersen. *Prosjektadministrasjon i systemutvikling. Aktiviteter i planleggningsfasen som påvirker suksess (In Norwegian)*. PhD thesis, ORAL, NTH, Trondheim, Norway, 1990.
28. W. J. Black, A. G. Sutcliffe, P. Loucopoulos, and P. J. Layzell. Translation between pragmatic software development methods. In H. K. Nichols and D. Simpson, editors, *ESEC '87 1st European Software Engineering Conference*, pages 357–365. Springer-Verlag, 1987.
29. B. I. Blum. A taxonomy of software development methods. *Communications of the ACM*, 37(11):82–94, November 1994.
30. R. Blumofe and A. Hecht. Executing real-time structured analysis specifications. *ACM SIGSOFT Software Engineering Notes*, 13(3):1–18, July 1988.
31. B. Boehm, P. Bose, E. Horowitz, and M. J. Lee. Software requirements negotiation and renegotiation aids: A theory-W based spiral model. In *17th International Conference on Software Engineering (ICSE'95)*, pages 243–253, Seattle, Washington, USA, April 23–30 1995.
32. B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72, May 1988.
33. D. P. Bogia. *Supporting Flexible, Extensible Task Descriptions in and Among Tasks*. PhD thesis, 1995.
34. C. Boldyref, E. L. Burd, and R. M. Hather. An evaluation of the state of the art for application management. In Müller and Georges [266], pages 161–169.
35. G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.
36. A. Borgida. Features of languages for the development of information systems at the conceptual level. *IEEE Software*, pages 63–72, January 1985.
37. A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements specification. *IEEE Computer*, 18(4):82–91, April 1985.
38. S. Bråten. *Dialogens vilkår i datasamfunnet (In Norwegian)*. Universitetsforlaget, 1983.

39. R. Brea. *Algebraic Specification Technologies in Object Oriented Programming Environments.* Springer-Verlag, 1991.
40. S. Brinkkemper. *Formalisation of Information Systems Modelling.* PhD thesis, University of Nijmegen, 1990. Thesis Publishers.
41. F. P. Brooks Jr. No silver bullet. Essence and accidents of software engineering. In H. J. Kugler, editor, *Information Processing '86*, pages 1069–1076. North-Holland, 1986.
42. M. Broy and P. Pepper. Program development as a formal activity. *IEEE Transactions on Software Engineering*, 7(1):14–22, January 1981.
43. T. Bryant and A. Evans. OO oversold: Those objects of obscure desire. *Information and Software Technology*, 36(1):35–42, January 1994.
44. J. A. Bubenko jr. Information modelling in the context of systems development. In *IFIP Conference*, 1980. Invited paper.
45. J. A. Bubenko jr. On concepts and strategies for requirements and information analysis. In *Information Modelling*, pages 125–169. Chartwell-Bratt Ltd., 1983.
46. J. A. Bubenko jr. Problems and unclear issues with hierarchical business activity and data flow modelling. Technical Report 134, SYSLAB, Stockholm, June 1988.
47. J. A. Bubenko jr, C. Rolland, P. Loucopoulos, and V. DeAntonellis. Facilitating fuzzy to formal requirements modelling. In *Proceedings of the First International Conference on Requirements Engineering (ICRE94)*, pages 154–157, Colorado Springs, USA, April 18-22 1994. IEEE Computer Society Press.
48. C. V. Bullen and J. L. Bennett. Groupware in practice: An interpretation of work experiences. In C. Dunlop and R. Kling, editors, *Computerization and Controversy : Value Conflicts and Social Choices*, pages 257–287. Academic Press, 1991.
49. M. Bunge. The metaphysics, epistemology, and methodology of levels. Elsevier, New York, 1969.
50. G. Burrel and G. Morgan. *Sociological Paradigms and Organizational Analysis.* Heinemann, 1979.
51. J. R. Cameron. An overview of JSD. *IEEE Transactions on Software Engineering*, 12(2):222–240, February 1986.
52. M. A. M. Capretz and M. Munro. Software configuration management issues in the maintenance of existing system. *Journal of Software Maintenance*, 6:1–14, 1994.
53. S. Carlsen. *Conceptual Modeling and Composition of Flexible Workflow Models.* PhD thesis, 1998.
54. S. Carlsen, J. Krogstie, A Sølvberg, and O. I. Lindland. Evaluating flexible workflow systems. Accepted at HICSS'30, 1997.
55. E. Carmel, R. D. Whitaker, and J. F. George. PD and joint application design: A transatlantic comparison. *Communications of the ACM*, 36(4):40–48, June 1993.
56. R. Carnap. *Meaning and Necessity.* University of Chicago Press, 1947.
57. C. Cauvet, C. Proix, and C. Rolland. ALECSI: An expert system for requirements engineering. In Andersen et al. [8], pages 31–49.
58. P. Chaiyasut, G. Shanks, and P. M. C. Swatman. A computational architecture to support conceptual data model reuse by analogy. In H. A. Müller and R. J. Norman, editors, *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE'95)*, pages 70–79, Toronto, Canada, July 10-14 1995. IEEE Computer Society Press.
59. B. Chandrasekaran, M. C. Tanner, and J. R. Josephson. Explaining control strategies in problem solving. *IEEE Expert*, 4(1):9–24, Spring 1989.

60. A.-M. Chang and T.-D. Han. Design of an argumentaion-based support system. In Nunamaker and Sprague [278].
61. P. B. Checkland. *Systems Thinking, Systems Practice*. John Wiley & Sons, 1981.
62. P. P. Chen. The entity-relationship model: Towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.
63. L. Chung. Dealing with security requirements during the development of information systems. In Rolland et al. [315], pages 234–251.
64. L. Chung, P. Katalagarianos, M. Marakakis, M. Mertikas, J. Mylopoulos, and Y. Vassiliou. From information systems requirements to designs: A mapping framework. *Information Systems*, 16(4):429–461, 1991.
65. P. M. Churchland. *A Neurocomputational Perspective*. MIT Press, 1989.
66. W. J. Clancey. The epistemology of a rule-based expert system - a framework for explanation. *Artificial Intelligence*, 20:215–251, 1983.
67. P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice-Hall, Englewood Cliffs, first edition, 1990.
68. D. Coleman, F. Hayes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.
69. D. L. Coleman and A. L. Baker. Deliberations on Kung's process interface modeling. *The Journal of Systems and Software*, pages 193–198, 1991.
70. J. Conklin and M. J. Begeman. gIBIS: A hypertext tool for exploratory policy discussion. *ACM Transactions on Office Information Systems*, 6(4):303–331, 1988.
71. J. L. Connell and L. B. Shafer. *Structured rapid prototyping - An Evolutionary Approach to Software Development*. Prentice Hall, 1989.
72. AMADEUS consortium. The AMADEUS Project: Final Report. ESPRIT report, June 1987.
73. R. J. Costello and D-B. Liu. Metrics for requirements engineering. *Journal of Systems and Software*, 29(1):39–63, April 1995.
74. T. G. Cummings and E. F. Huse. *Organization Development and Change*. West, 1989.
75. B. Dahlbom. The idea that reality is socialy constructed. In Floyd et al. [121], pages 101–126.
76. R. Dale, C. Mellish, and M. Zock, editors. *Current Research in Natural Language Generation*. Academic Press Limited, 1990.
77. H. Dalianis. A method for validating a conceptual model by natural language discourse generation. In Loucopoulos [242], pages 425–444.
78. R. Darimont and A. van Lamsweerde. Formal requirement patterns for goal-driven requirements elaboration. In *Proceedings of SIGSOFT'96*, pages 179–190, 1996.
79. A. M. Davis. A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, 31(9):1098–1115, September 1988.
80. A. M. Davis. *Software Requirements Analysis & Specification*. Prentice-Hall, 1990.
81. A. M. Davis, S. Overmeyer, K. Jordan, J. Caruso, F. Dandashi, A. Dinh, G. Kincaid, G. Ledeboer, P. Reynolds, P. Sitaram, A. Ta, and M. Theofanos. Identifying and measuring quality in a software requirements specification. In *Proceedings of the First International Software Metrics Symposium*, pages 141–152, 1993.
82. R. Davis and J. King. An overview of production systems. In E. W. Elcock and D. Michie, editors, *Machine Intelligence*, pages 300–332. 1977.

83. G. de Michelis and M. A. Grasso. Situating conversations within the language/action perspective: The Milan conversation model. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, pages 89–100, Chapel Hill, North Carolina, USA, October 22-26 1994.
84. G-J. de Vrede. Support for collaborative design: Animated electronic meetings. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences (HICCS'97): Volume II Information Systems- Collaboration Systems and Technology*, pages 376–385. IEEE Computer Society Press, 1997.
85. J. K. Debenham and G. M. McGrath. The description in logic of large commercial data bases: A methodology put to the test. In *Proceedings of the Fifth Australian Computer Science Conference*, pages 12–21, 1982.
86. S. M. Dekleva. Delphi study of software maintenance problems. In *Proceedings of the Conference on Sofware Maintenance (CSM'92)*, pages 10–17, 1992.
87. S. M. Dekleva. The influence of the information systems development approach on maintenance. *MIS Quarterly*, pages 355–372, September 1992.
88. H. S. Delugach. Specifying multiple-viewed software requirements with conceptual graphs. *Journal of Systems and Software*, 19:207–224, 1992.
89. P. J. Denning. What is software quality. *Communications of the ACM*, 35(1):13–15, January 1992.
90. Y. Deville. *Logic Programming - Systematic Program Development.* International Series in Logic Programming. Addison Wesley, 1990.
91. J. Dietz. Integrating management of human and computer resources in task processing organizations: A conceptual view. In Nunamaker and Sprague [277], pages 723–733.
92. J. L. G. Dietz and G. A. M. Widdershoven. A comparison of the lingusitic theories of Searle and Habermas as a basis for communication supporting systems. In R. P. van Riet and R. A. Meersman, editors, *Linguistic Instruments in Knowledge Engineering*, pages 121–130. Elsevier, 1992.
93. F. Dignum, T. Kemme, W. Kreuzen, R. Weigand, and R. P. van de Riet. Constraint modelling using a conceptual prototyping language. *Data & Knowledge Engineering*, (2):213–254, 1987.
94. F. Dignum and H. Weigand. Communication and deontic logic. In R. Wieringa and R. Feenstra, editors, *Working papers of the International Workshop on Information Systems - Correctness and Reuseability (IS-CORE '94)*, 1994.
95. E. Downs, P. Clare, and I. Coe. *Structured Systems Analysis and Design Method: Application and Context.* Prentice Hall, 1988.
96. D. R. Dowty et al. *Introduction to Montague Semantic.* Reidel, 1981.
97. K. G. Doyle, J. R. G. Wood, and A. T. Wood-Harper. Soft systems and systems engineering: On the use of conceptual models in information system development. *Journal of Information Systems*, 3(3):187–198, 1993.
98. E. Dubois, P. Du Bois, and M. Petit. Elicitating and formalising requirements for C.I.M. information systems. In Rolland et al. [315], pages 253–274.
99. E. Dubois, P. Du Bois, and M. Petit. ALBERT: An agent-oriented language for building and elicitating requirements for real-time systems. In Nunamaker and Sprague [277], pages 713–722, Volume 4.
100. S. Easterbrook. Domain modelling with hierarchies of alternative viewpoints. In *Proceedings of the IEEE International Symposium on Requirements Engineering (RE'93)*, pages 65–72, San Diego, USA, January 4-6 1993.
101. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer, 1985.
102. M. Elden and M. Levin. Co-generative learning. Bringing participation into action research. In W. F. Whyte, editor, *Participative Action Research.* Sage, 1991.

103. C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware, some issues and experiences. *Communications of the ACM*, 34(1):39–58, January 1991.
104. H. C. Ellis and R. R. Hunt. *Fundamentals of Cognitive Psychology.* Brown and Benchmark, Madison, Wisconsin, 1993. 5th edition.
105. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., 1989.
106. D. W. Embley, R. B. Jackson, and S. N. Woodfield. OO system analysis: Is it or isn't it? *IEEE Software*, 12(3):19–33, July 1995.
107. M. Emery. The theory and practice of search conferences. In *Proceedings from the Einar Thorsrud Memorial Symposium and Workshop*, Oslo, Norway, June 15-19 1987.
108. F. Fabbrini, M. Fusani, V. Gervasi, S. Gnesi, and S. Ruggieri. Achieving quality in natural language requirements. In *Proceedings of the 11th International Software Quality Week (QW98)*, San Francisco, USA, May 26-29 1998.
109. E. D. Falkenberg, W. Hesse, P. Lindgreen, B. E. Nilsson, J. L. Han Oei, C. Rolland, R. K. Stamper, F. J. M. Van Assche, A. A. Verrijn-Stuart, and K. Voss. FRISCO: A Framework of Information System Concepts. Technical report, IFIP WG 8.1, December 1997.
110. E. D. Falkenberg, R. van der Pols, and Th. P. van der Weide. Understanding process structure diagrams. *Information Systems*, 16(4):417–428, 1991.
111. B. A. Farshchian, J. Krogstie, and A. Sølvberg. Integration of user interface and conceptual modeling. In *Proceedings of the ERCIM workshop: Towards User Interfaces for All: Current efforts and future trends*, Heraklion, Crete, Greece, October 30-31 1995. ICS-FORTH.
112. M. S. Feather. Mappings for Rapid Prototyping. *ACM SIGSOFT Software Engineering Notes*, 7(5):17–24, December 1982.
113. M. S. Feather. Requirements engineering - Getting right from wrong. In A. Van Lamsweerde and A. Fugetta, editors, *ESEC'91 — 3rd European Software Engineering Conference*, pages 485–488, Milan, Italy, October 1991. Springer-Verlag.
114. M. S. Feather. Requirement reconnoitering at the juncture of domain and instance. In *Proceedings of the IEEE International Symposium on Requirements Engineering (RE'93)*, pages 73–76, San Diego, USA, January 4-6 1993.
115. A. Feller and R. Rucker. Meta-modeling systems analysis primitives. In *Conceptual Structures: Current Research and Practice*, chapter 10, pages 201–220. Ellis Horwood, 1992.
116. J. Fiadeiro et al. Describing and structuring objects for conceptual schema development. *P. Loucopoulas (eds):* Conceptual Modelling, Databases and CASE, 1991.
117. N. V. Findler, editor. *Associative Networks: Representation and Use of Knowledge by Computer.* Academic Press, 1979.
118. F. Flores, M. Graves, B. Hartfield, and T. Winograd. Computer systems and the design of organizational interaction. *ACM Transactions on Office Information Systems*, 6(2):153–172, April 1988.
119. C. Floyd. A systematic look at prototyping. In R. Budde et al., editor, *Approaches to Prototyping*, pages 1–18, Berlin, 1984. Springer-Verlag.
120. C. Floyd, F-M. Reisin, and G. Schmidt. STEPS to software development with users. In C. Ghezzi and J. A. McDermid, editors, *2nd European Software Engineering Conference (ESEC'89)*, pages 48–63, University of Warwick, Coventry, England, September 1989.
121. C. Floyd, H. Züllighoven, R. Budde, and R. Keil-Slawik, editors. *Software Development and Reality Construction.* Springer Verlag, 1991.

122. W. Frakes and C. Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2):415–435, June 1996.
123. C. Francalanci and B. Pernici. View integration: A survey of current developments. Technical Report 93-053, Politecnico de Milano, Milan, Italy, 1993.
124. E. Francik, S. E. Rudman, D. Cooper, and S. Levine. Putting innovation to work: Adoption strategies for multimedia communication systems. *Communications of the ACM*, 34(12):52–63, December 1991.
125. M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Informal and formal requirements specification languages: Bridging the gap. *IEEE Transactions on Software Engineering*, 17(5):454–466, May 1991.
126. Personal communication with the FRISCO task group, March 1995.
127. A. Gal, G. Lapalme, P. Saint-Dizier, and H. Somers. *Prolog for Natural Language Processing*. John Wiley & Sons, New York, USA, 1991.
128. H. Gallaire, J. Minker, and J. Nicolas. Logic and databases: A deductive approach. *Computing Surveys*, 16(2):153–185, June 1984.
129. C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques*. Prentice-Hall, 1979.
130. H. Garfinkel. *Studies in Ethnomethodology*. Prentice Hall, 1967.
131. D. Garlan, C. Krueger, and B. Staudt. A structural approach to the maintenance of structure-oriented environments. *ACM SIGPLAN Notices*, 22(1):160–170, January 1987.
132. D. G. Gause and G. M. Weinberg. *Exploring Requirements: Quality before Design*. 1989.
133. M. R. Genesereth and S. T. Ketchpel. Software agents. *Communication of the ACM*, 37(7):48–53, July 1994.
134. M. R. Genesereth and J. N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufman, 1988.
135. A. Gill. *Applied Algebra for the Computer Sciences*. Prentice-Hall, 1976.
136. R. Gjersvik. *The Construction of Information Systems in Organization: An Action Research Project on Technology, Organizational Closure, Reflection, and Change*. PhD thesis, ORAL, NTH, Trondheim, Norway, 1993.
137. R. L. Glass. We have lost our way. *Journal of Systems and Software*, 18(2):111–112, May 1992.
138. R. L. Glass and I. Vessey. Contemporary application-domain taxonomies. *IEEE Software*, 12(3):63–76, July 1995.
139. B. C. Glasson. Model of system evolution. *Information and Software Technology*, 31(7):351–356, September 1989.
140. J. A. Goguen. Parameterized Programming. *IEEE Transactions on Software Engineering*, SE-12(5):528–543, September 1984.
141. J. A. Goguen and C. Linde. Techniques for requirements elicitation. In *Proceedings of the IEEE International Symposium on Requirements Engineering (RE'93)*, pages 152–164, 1993.
142. G. Goldkuhl and K. Lyytinen. A language action view of information systems. In *Proceedings of the International Conference on Information Systems (ICIS'82)*, pages 13–29. 1982.
143. G. Goldkuhl and K. Lyytinen. Information systems specification as rule reconstruction. In Th. M. A. Bemelmans, editor, *Beyond Productivity: Information Systems Development for Organizational Effectiveness*, pages 79–94. North-Holland, 22-24 August 1983.
144. H. Gomaa. The impact of rapid prototyping on specifying user requirements. *ACM SIGSOFT - Software Engineering Notes*, 8(2):17–28, April 1983.

145. H. Gomaa and D. B. H. Scott. Prototyping as a tool in the specification of user requirements. In *The Proceedings of the 5th International Conference on Software Engineering*, pages 333–342, 1981.
146. E. S. Greenberg. The consequences of worker participation: A clarification of the theorethical litterature. *Social Science Quarterly*, 56(2), 1975.
147. Grobstein. Hierarchical order and neogenesis. In Pattee [299].
148. HOOD Working Group. HOOD reference manual. Technical Report WME/89-173/JB, European Space Agency, September 1989.
149. E. G. Guba and Y. S. Lincoln. *Fourth Generation Evaluation*. Sage, 1989.
150. J. A. Gulla. Code generation in PPP - documentation and evaluation. Internal report, IDT, NTH, Trondheim, Norway, 1991.
151. J. A. Gulla. *Explanation Generation in Information Systems Engineering*. PhD thesis, IDT, NTH, Trondheim, Norway, September 1993.
152. J. A. Gulla, O. I. Lindland, and G. Willumsen. PPP - An integrated CASE environment. In Andersen et al. [8], pages 194–221.
153. M. R. Gustafsson, T. Karlsson, and J. A. Bubenko jr. A declarative approach to conceptual information modelling. In Olle et al. [285], pages 93–142.
154. J. Habermas. *The Theory of Communicative Action*. Beacon Press, 1984.
155. J. Hagelstein. A declarative approach to information systems requirements. *Knowledge Based Systems*, 1(4):211–220, 1988.
156. J. Hagelstein and A. Rifaut. A comparison of semantic models for collections. Technical report, Philips Research Lab, Brussels, Belgium, 1987.
157. U. Hahn, M. Jarke, and T. Rose. Group work in software projects: Integrated conceptual models and collaboration tools. In S. Gibbs and A. A. Verrijn-Stuart, editors, *Multi-User Interfaces and Applications: Proceedings of the IFIP WG 8.4 Conference on Multi-User Interfaces and Applications*, pages 83–102. North-Holland, 1990.
158. G. F. Håland. SIMSPEC - Simulating a PPP specification. Master's thesis, IDT, NTH, Trondheim, Norway, 1991.
159. D. P. Hale, D. A. Haworth, and S. Sharpe. Empirical software maintenance studies during the 1980s. In *Proceedings of the Conference on Software Maintenance (CSM'90)*, pages 118–123. IEEE Computer Society Press, 1990.
160. M. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems*, 6(3), September 1981.
161. D. Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, (8):231–274, 1987.
162. D. Harel. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, January 1992.
163. D. Harel and E. Gery. Executable object modelling with statecharts. In *18th International Coference on Software Engineering (ICSE'96)*, pages 246–257, Berlin, Germany, March 25-29 1996.
164. D. Harel, H. Lachover, A. Naamed, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
165. W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 411–428, 1993.
166. F. Heller. Participation and competence: A necessary relationship. In R. Russel and V. Rus, editors, *International Handbook of Participation in Organizations*, pages 265–281. 1991.

167. P. Henderson. Functional programming, formal specification, and rapid prototyping. *IEEE Transactions on Software Engineering*, SE-12(2):241–250, February 1986.
168. J. Hewett and T. Durham. CASE: The Next Steps. Technical report, OVUM, 1989.
169. R. A. Hirschheim. A participative approach to implementing office automation. In *Proceedings from the Joint International Symposium on Information Systems*, pages 306–329, Sydney, Australia, April 1984.
170. R. A. Hirschheim and H. K. Klein. Four paradigms of information systems development. *Communications of the ACM*, 32(10):pages 1199–1216, October 1989.
171. K. K. Holgeid. Utvikling og vedlikehold av it-systemer i norske bedrifter (in norwegian). Master's thesis, IFI, UIO, Oslo, Norway, April 1999.
172. P. Holm. The COMMODIOUS method: Communication modelling as an aid to illustrate the organizational use of software. In *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering (SEKE'94)*, pages 10–19, Jurmala, Latvia, June 21-23 1994. Knowledge Systems Institute.
173. G. M. Høydalsvik and G. Sindre. On the purpose of object-oriented analysis. In A. Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 240–255. ACM Press, September 1993.
174. R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.
175. R. Hull and R. King. A tutorial on semantic database modellig. In *Research Foundations in Object-oriented and Semantic Database Systems*. Prentice Hall, 1990.
176. J. Iivari. Hierarchical spiral model for information system and software development. Part 1: Theoretical background. *Information and Software Technology*, 32(6):386–399, July/August 1990.
177. J. Iivari. Hierarchical spiral model for information system and software development. Part 2: Design process. *Information and Software Technology*, 32(7):450–458, September 1990.
178. J. Iivari. Object-orientation as structural, functional and behavioural modelling: A comparison of six methods for object-oriented analysis. *Information and Software Technology*, 37(3):155–163, 1995.
179. J. Iivari, K. Lyytinen, and M. Rossi, editors. *Proceedings of the 7th International Conference on Advanced Information Systems Engineering (CAiSE'95)*, Jyväskylä, Finland, June 12-16 1995. Springer Verlag.
180. H. Ishii and N. Miyake. Toward an open shared workspace: Computer and video fusion approach of teamworkstation. *Communications of the ACM*, 34(12):36–50, December 1991.
181. M. L. Jaccheri and R. Conradi. Techniques for process model evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12):1145–56, December 1993.
182. B. E. Jacobs. On database logic. *Journal of the ACM*, 29(2):310–332, April 1982.
183. S. Jacobs, M. Jarke, and K. Pohl. Report on the first international IEEE symposium on requirements engineering (RE'93) San Diego, January 4–6 1993. *Automated Software Engineering*, 1(1):129–132, 1994.
184. I. Jacobson et al. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, Reading, MA, 1992.

185. James Martin Associates PLC. *IEM - Information Engineering Methodology - Introduction*, 1987.
186. M. Jarke, J. A. Bubenko jr, C. Rolland, A. Sutcliffe, and Y. Vassiliou. Theories underlying requirements engineering: An overview of NATURE at genesis. In *Proceedings of the IEEE International Symposium on Requirements Engineering (RE'93)*, pages 19–31, 1993.
187. M. Jarke, J. Mylopoulos, J. W. Schmidt, and Y. Vassiliou. DAIDA: An environment for evolving information systems. *ACM Transactions on Information Systems*, 10(1):1–50, 1992.
188. P. Johanneson. Schema transformations as an aid in view integration. In Rolland et al. [315], pages 71–92.
189. W. L. Johnson and M. S. Feather. Using evolution transformations to construct specifications. In M. R. Lowry and R. D. McCartney, editors, *Automating Software Design*, pages 65–91. The MIT Press, California, USA, 1991.
190. W. L. Johnson, M. S. Feather, and D. R. Harris. Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering*, 18(10):853–869, October 1992.
191. A. J. I. Jones and I. Pörn. Ideality, sub-ideality and deontic logic. *Synthese*, 65:275–290, 1985.
192. A. J. I. Jones and I. Pörn. "ought" and "must". *Synthese*, 66:89–93, 1986.
193. J. R. Jørgensen, O. T. Kogstad, and H. Nilsen. Rapid prototyping of user interfaces in the I-CASE environment PPP. Master's thesis, IDT, NTH, Trondheim, Norway, 1991.
194. M. Jørgensen. *Empirical studies of Software Maintenance*. PhD thesis, Department of Informatics, University of Oslo, Oslo, Norway, 1994.
195. M. Jørgensen and A. Maus. A case study of software maintenance tasks. In *Proceedings of Norsk Informatikk Konferanse 1993 (NIK'93)*, pages 101–112, Halden, Norway, 1993.
196. R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-oriented specification of information systems: The T_{ROLL} language. Technical report, Technische Universität Braunschweig, 1991.
197. R. Jungclaus, G. Saake, and C. Sernadas. Formal specification of object systems. In S. Abramsky and T. Maibaum, editors, *Proceedings of TAPSOFT'91*, pages 60–82, Volume 2, Brighton, UK, 1991. Springer Verlag (LNCS 651).
198. J. Kalita. Automatically generating natural language reports. *International Journal of Man-Machine Studies*, 30(4):399–423, April 1989.
199. E.-A. Karlsson (ed.). *Software Reuse: A Holistic Approach*. John Wiley & Sons, 1995.
200. M. Keil and E. Carmel. Customer-developer links in software development. *Communications of the ACM*, 38(5), May 1995.
201. S. Khoshafian and R. Abnous. *Object Orientation: Concepts, Languages, Databases, User interfaces*. Wiley, 1990.
202. S. Khosla, T. S. E. Maibaum, and M. Sadler. Database specification. In T. B. Steel and R. Meersman, editors, *Proceedings of the IFIP Conference on Database Semantics (DS-1)*, 1986.
203. H. Klein and K. Lyytinen. Towards a new understanding of data modelling. In Floyd et al. [121], pages 203–217.
204. M. D. Konrad. *Functional Prototyping with Proto*, chapter 12, pages 378–398. Van Nostrand Reinhold, 1989. Edited by Peter A. Ng and Raymond T. Yeh.
205. W. Kozaczynski and A. Kuntzmann-Combelles. What it takes to make OO work. *IEEE Software*, 10(1):20–23, January 1993.
206. J. Krogstie. Conceptual modeling in Tempora. Master's thesis, IDT, NTH, Trondheim, Norway, 1991.

207. J. Krogstie. *Conceptual Modeling for Computerized Information Systems Support in Organizations.* PhD thesis, IDT, NTH, Trondheim, Norway, November 21 1995.
208. J. Krogstie. Goal-oriented modeling of information systems. In *Proceedings of the Seventh International Conference on Computing and Information (ICCI'95)*, pages 983–1007, Peterborough, Canada, July 5–8 1995.
209. J. Krogstie. On the distinction between functional development and functional maintenance. *Journal of Software Maintenance*, 7:383–403, 1995.
210. J. Krogstie, O. I. Lindland, and G. Sindre. Defining quality aspects for conceptual models. In *Proceedings of the IFIP8.1 working conference on Information Systems Concepts (ISCO3): Towards a Consolidation of Views*, Marburg, Germany, March 28-30 1995.
211. J. Krogstie, O. I. Lindland, and G. Sindre. Towards a deeper understanding of quality in requirements engineering. In Iivari et al. [179], pages 82–95.
212. J. Krogstie, P. McBrien, R. Owens, and A. H. Seltveit. Information systems development using a combination of process and rule based approaches. In Andersen et al. [8], pages 319–335.
213. J. Krogstie and G Sindre. Utilizing deontic operators in information systems specification. Issued to Requirements Engineering Journal.
214. J. Krogstie and G. Sindre. Extending a temporal rule language with deontic operators. In *Proceedings from the 6th International Conference on Software Engineering and Knowledge Engineering (SEKE'94)*, pages 314–321. IEEE, June 21-23 1994.
215. J. Krogstie and A. Sølvberg. Software maintenance in Norway: A survey investigation. In Müller and Georges [266], pages 304–313. Received "Best Paper Award".
216. C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.
217. C. H. Kung. A temporal framework for database specification and verification. In *Proceedings of the tenth International Conference on Very Large Data Bases (VLDB'84)*, pages 91–99, Singapore, August 27-31 1984.
218. C. H. Kung. *A Temporal Framework for Information Systems Specification and Verification.* PhD thesis, IDT, NTH, Trondheim, Norway, 1984.
219. C. H. Kung. Object subclass hierarchy in SQL: A simple approach. *Communication of the ACM*, 33(7):117–125, 1990.
220. C. H. Kung. Process interface modeling and consistency checking. *The Journal of Systems and Software*, pages 185–191, 1991.
221. M. Kyng. Designing for cooperation: Cooperating for design. *Communications of the ACM*, 34(12):65–73, December 1991.
222. V. Lalioti. Animation for validation of business system specification. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences (HICCS'97): Volume II Information Systems- Collaboration Systems and Technology.* IEEE Computer Society Press, 1997.
223. B. Langefors. *Theoretical Analysis of Information Systems.* Studentliteratur, Auerbach, first edition, 1973.
224. P. J. Layzell and L. Macauley. An investigations into software maintenance - perception and practices. *Software Maintenace: Research and Practice*, 6:105–119, 1994.
225. R.-J. Lea and C.-G. Chung. Rapid prototyping from structured analysis: Executable specification approach. *Information and Software Technology*, 32(9):589–597, November 1990.
226. C. H. LeDoux and D. S. Parker Jr. Saving traces for ADA debugging. In *Ada In Use, Proceedings of The Ada International Conference*, pages 97–108, 1985.

227. S. Lee and S. Sluizer. An executable language for modeling simple behavior. *IEEE Transactions on Software Engineering*, 17(6):527–543, June 1991.
228. M. Lefering. An incremental integration tool between requirements engineering and programming in the large. In *Proceedings of the IEEE International Symposium on Requirements Engineering (RE'93)*, pages 82–89. IEEE Computer Society Press, 1993.
229. J. C. S. P. Leite and P. A. Freeman. Requirements validation through viewpoint resolution. *IEEE Transactions on Software Engineering*, 17(12):1253–1269, December 1991.
230. H. R. Lewis. Cycles of unifiability and decidability by resolution. Technical report, Aiken Computation Laboratory, Harvard University, 1979.
231. H. R. Lewis. *Unsolvable Classes of Quantificational Formulae*. Addison-Wesley, 1979.
232. P. J. Lewis. Linking soft systems methodology with data-focused information systems development. *Journal of Information Systems*, 3:169–186, 1993.
233. X. Li. What's so bad about rule-based programming? *IEEE Software*, 8(5):103,105, September 1991.
234. B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison Wesley, 1980.
235. P. Lindgren ed. A framework of information systems concepts. Technical report, FRISCO, May 1990.
236. O. I. Lindland. *A Prototyping Approach to Validation of Conceptual Models in Information Systems Engineering*. PhD thesis, IDT, NTH, Trondheim, Norway, May 1993.
237. O. I. Lindland and J. Krogstie. Transformations in CASE tools - A compiler view. In H. Y. Lee, T. Reid, and S. Jarzabek, editors, *Proceedings of the 6th International Workshop on Computer-Aided Software Engineering (CASE'93)*, pages 287–296, Singapore, July 1993. IEEE Computer Society Press.
238. O. I. Lindland and J. Krogstie. Validating conceptual models by transformational prototyping. In Rolland et al. [315], pages 165–183.
239. O. I. Lindland, G. Sindre, and A. Sølvberg. Understanding quality in conceptual modelling. *IEEE Software*, pages 42–49, April 1994.
240. O. I. Lindland, G. Willumsen, J. A. Gulla, and A. Sølvberg. Prototyping in transformation-based CASE environments. In *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering (SEKE'93)*, pages 696–603, Hotel Sofitel, San Francisco Bay, USA, 1993. Knowledge Systems Institute.
241. J. Lingat, P. Colignon, and C. Rolland. Rapid application prototyping - the PROQUEL language. In F. Bancilhon and D. J. DeWitt, editors, *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB'14)*, pages 206–217, 1988.
242. P. Loucopoulos, editor. *Proceedings of the Fourth International Conference on Advanced Information Systems Engineering (CAiSE'92)*. Springer-Verlag, May 1992.
243. P. Loucopoulos, P. McBrien, F. Schumacker, B. Theodoulidis, V. Kopanas, and B. Wangler. Integrating database technology, rule-based systems and temporal reasoning for effective information systems: The TEMPORA paradigm. *Journal of Information Systems*, 1:129–152, 1991.
244. P. H. Loy. A comparison of object-oriented and stuctured development methods. *ACM SIGSOFT Software Engineering Notes*, 15(1):44–48, January 1990.
245. M. D. Lubars. A general design representation. Technical Report STP-066-89, Microelectronics and computer technology corporation (MCC), 1989.

246. M. Lundeberg. The ISAC approach to specification of information systems and its application to the organization of an IFIP working conference. In *Information Systems Design Methodologies: A Comparative Review*, pages 173–234. North-Holland, 1982.
247. J. Lyons. *Introduction to Theoretical Linguistics*. Cambridge University Press, 1968.
248. K. Lyytinen. A taxonomic perspective of information systems development: Theoretical constructs and recommendations. In R. J. Boland Jr and R. A. Hirschheim, editors, *Critical Issues in Information Systems Research*, chapter 1, pages 3–41. John Wiley & Sons, 1987.
249. K. Lyytinen, P. Kerola, J. Kaipala, S. Kelly, J. Lehto, H. Liu, P. Marttiin, H. Oinas-Kukkonen, J. Prihonen, M. Rossi, K. Smolander, V-P. Tahvanainen, and J-P. Tolvanen. MetaPHOR: Metamodelling, principles, hypertext, objects, and repositories. Technical Report TR-7, University of Jyväskylä, Department of Computer Science and Information Systems, Jyväskylä, Finland, November 1994.
250. L. Macauley. Requirements capture as a cooperative activity. In *Proceedings of the First Symposium on Requirements Engineering (RE'93)*, pages 174–181, 1993.
251. N. A. Maiden and A. G. Sutcliffe. Exploiting reusable specifications through analogy. *Communications of the ACM*, 35(4):55–64, April 1992.
252. W. C. Mann and S. A. Thompson. Rhetorical structure theory: Description and construction of text structures. In G. Kempen, editor, *Natural Language Generation: New Results in Artificial Intelligence, Psychology, and Linguistics*, chapter 7, pages 85–95. Martinus Nijhoff Publishers, 1987.
253. M. A. Marsan et al, editor. *Proceeding of the International workshop on Timed Petri Nets*, Torino, Italy, 1985. IEEE Computer Society Press.
254. P. McBrien, M. Niezette, D. Pantazis, A. H. Seltveit, U. Sundin, B. Theodoulidis, G. Tziallas, and R. Wohed. A rule language to capture and model business policy specifications. In Andersen et al. [8], pages 307–318.
255. P. Mcbrien and A. H. Seltveit. Coupling process models and business rules. In Sølvberg et al. [349], pages 201–217.
256. P. McBrien, A. H. Seltveit, and B. Wangler. An entity-relationship model extended to describe historical information. In *Proceedings of CISMOD'92*, Bangalore, India, July 1992.
257. D. McCracken and M. Jackson. Life cycle concept considered harmful. *ACM SIGSOFT Software Engineering Notes*, 7(2):29–32, April 1982.
258. M. P. McDonald. Quality function deployment. introducing product development into the systems development process. In *Seventh Symposium on Quality Function Deployment*, Novi, Michigan, June 1995.
259. C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–623, December 1989.
260. R. Medina-Mora, T. Winograd, R. Flores, and F. Flores. The action workflow approach to workflow management technology. In *Proceedings of CSCW'92*, 1992.
261. M. D. Mesarović et al. *Theory of Hierarchical, Multilevel, Systems*. Academic Press, 1970.
262. B Meyer. Reality: A cousin twice removed. *IEEE Computer*, 29(7):96–97, July 1996.
263. D. L. Moody and G. G. Shanks. What makes a good data model? Evaluating the quality of entity relationship models. In *Proceedings of the 13th International Conference on the Entity-Relationship Approach (ER'94)*, pages 94–111, Manchester, England, 1994.

264. R Motschnig-Pitrik. The semantics of parts versus aggregates in data/knowledge modeling. In Rolland et al. [315].
265. B. Moulin and D. Rousseau. SACD: A system for acquiring knowledge from regulatory texts. *Computers and Electrical Engineering*, 20(2):131–149, 1994.
266. H. A. Müller and M. Georges, editors. *Proceedings of the International Conference on Software Maintenance (ICSM'94)*. IEEE COmputer Society Press, September 19-23 1994.
267. E. Mumford. Designing human systems - the ETHICS method. Technical report, Manchester Business School, Cheshire, England, 1983.
268. E. Mumford. Participation - from Aristotle to today. In Th. M. A. Bemelmans, editor, *Beyond Productivity: Information Systems Development for Organizational Effectiveness*, pages 95–104. North-Holland, 22-24 August 1983.
269. J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. TELOS: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.
270. J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, June 1992.
271. J. Mylopoulos et al. A language facility for designing database intensive applications. *ACM Transactions on Database Systems*, 5(2), June 1980.
272. R. Neches, W. R. Swartout, and J. D. Moore. Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Transactions on Software Engineering*, 11(11):1337–1351, November 1985.
273. G. M. Nijssen and T. A. Halpin. *Conceptual Schema and Relational Database Design*. Prentice Hall, 1989.
274. C. Niskier and T. Maibaum. A pluralistic knowledge-based approach to software specification. In C. Ghezzi and J. A. McDermid, editors, *ESEC'89 2nd European Software Engineering Conference*, pages 411–423. Springer Verlag, 1989.
275. I. Nonaka. A dynamic theory of organizational knowledge creation. *Organization Science*, 5(1):14–37, 1994.
276. W. Nöth. *Handbook of Semiotics*. Indiana University Press, 1990.
277. J. F. Nunamaker and R. H. Sprague, editors. *Proceedings of the Twenty-seventh Annual Hawaii International Conference on Systems Sciences (HICCS'27)*, Maui, Hawaii, US, January 4–7 1994. IEEE Computer Society Press.
278. J. F. Nunamaker and R. H. Sprague, editors. *Proceedings of the Twenty-eight Annual Hawaii International Conference on Systems Sciences (HICCS'28)*, Maui, Hawaii, US, January 1995. IEEE Computer Society Press.
279. B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, October 1994.
280. J. L. H. Oei. A meta model transformation approach towards harmonization in information system modelling. In *Proceedings of the IFIP8.1 working conference on Information Systems Concepts (ISCO3): Towards a Consolidation of Views*, Marburg, Germany, March 28-30 1995.
281. A. Olivé. A comparison of the operational and deductive approaches to conceptual information systems modelling. In H. J. Kugler, editor, *Information Processing '86*. North-Holland, 1986.
282. A. Olivé. On the design and implementation of information systems from deductive conceptual models. In *Proceedings of the 14th VLDB*, 1989.

283. T. W. Olle, J. Hagelstein, I. G. MacDonald, C. Rolland, H. G. Sol, F. J. M. van Assche, and A. A. Verrijn-Stuart. *Information Systems Methodologies.* Addison-Wesley, 1988.
284. T. W. Olle, J. Hagelstein, I. G. Macdonald, C. Rolland, H. G. Sol, F. J. M. van Assche, and A. A. Verrijn-Stuart. *Information Systems Methodologies - A Framework for Understanding.* Addison-Wesley, 1991.
285. T. W. Olle, H. G. Sol, and A. A. Verrijn-Stuart, editors. *Information Systems Design Methodologies: A Comparative Review.* North-Holland, 1982.
286. A. L. Opdahl. *Performance Engineering during Information Systems Development.* PhD thesis, IDT, NTH, Trondheim, Norway, 1992.
287. A. L. Opdahl and G. Sindre. A taxonomy for real-world modelling concepts. *Information Systems*, 19(3):229–241, April 1994.
288. A. L. Opdahl and G. Sindre. Facet models for problem analysis. In Iivari et al. [179], pages 54–67.
289. A. L. Opdahl and G. Sindre. Representing real-world processes. In Nunamaker and Sprague [278].
290. A. L. Opdahl and G. Sindre. Facet modeling: An approach to flexible and integrated conceptual modeling. *Information Systems*, 22(5):291–323, 1997.
291. J. W. Orlikowski and D. C. Gash. Technological frames: Making sense of information technology in organizations. *ACM Transactions on Information Systems*, 12(2):174–207, 1994.
292. OVUM evaluates CASE products. Technical report, OVUM, 1997.
293. R. P. Owens and P. McBrien. TEQUEL: the TEMPORA execution language. Technical Report E2469/IC/3.1/2/3, Imperial College, London, England, 1990.
294. P. Pagin. *Ideas for a Theory of Rules.* PhD thesis, Stockholm University, Stockholm, Sweden, 1987.
295. C. L. Paris. Generation and explanation: Building an explanation facility for the explainable expert systems framework. In Paris et al. [296], chapter 2, pages 49–82.
296. C. L. Paris, W. R. Swartout, and W. C. Mann, editors. *Natural Language Generation in Artificial Intelligence and Computational Linguistics.* Kluwer Academic Publishers, 1991.
297. Y. Park and D. Ramjisingh. Software component base for reuse in functional program development. In *Proceedings of the Seventh International Conference on Computing and Information (ICCI'95)*, pages 1022–1039, July 5–8 1995.
298. J. Parsons and Y. Wand. Choosing classes in conceptual modeling. *Communications of the ACM*, 40(6):63–69, June 1997.
299. H. H. Pattee, editor. *Hierarchy Theory.* Braziller, 1973.
300. M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber. Capability maturity model, version 1.1. *IEEE Software*, 10(4):pages 18–27, July 1993.
301. J. Peckham and F. Maryanski. Semantic data models. *ACM Computing Surveys*, 20(3):pages 153–190, September 1988.
302. B. Pernici and C. Rolland. Automatic tools for designing office information systems (TODOS). Research report 813, ESPRIT, 1990.
303. M. Petre. Why looking isn't always seeing. Readership skills and graphical programming. *Communications of the ACM*, 38(6):pages 33–44, June 1995.
304. C. A. Petri. Kommunikation mit automaten (In German). *Schriften des Rheinisch-Westfalischen Institut für Instrumentelle Mathematik an der Universität Bonn*, (2), 1962.
305. K. Pohl. The three dimensions of requirements engineering. In Rolland et al. [315], pages 275–292.
306. K. Pohl. The three dimensions of requirements engineering: A framework and its applications. *Information Systems*, 19(3):243–258, April 1994.

307. W. D. Potter and R. P. Trueblood. Traditional, semantic and hyper-semantic approaches to data modeling. *IEEE Computer*, 21(6):53–63, June 1988.
308. R. Prieto-Diaz. Status report: Software reuseability. *IEEE Software*, pages 61–66, May 1993.
309. W. v. O. Quine. *Set Theory and its Logic*. Belknap, Cambridge, Massachusets, 1963.
310. M. R. Raabel. User interface specification language for PPP. Master's thesis, IDT, NTH, Trondheim, Norway, 1993.
311. B. Ramesh and M. Edwards. Supporting systems development by capturing deliberations during requirements engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510, June 1992.
312. T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects*. Manning/Prentice Hall, 1995.
313. C. Rich and R. C. Waters. Automatic programming: Myths and prospects. *IEEE Computer*, 22(7):40–51, August 1988.
314. H. Rittel. On the planning crisis: Systems analysis of the first and second generations. *Bedriftsøkonomen*, 34(8), 1972.
315. C. Rolland, F. Bodart, and C. Cauvet, editors. *Proceedings of the 5th International Conference on Advanced Information Systems Engineering (CAiSE'93)*, Paris, France, June 8-11 1993. Springer Verlag.
316. C. Rolland and C. Proix. A natural language approach for requirements engineering. In Loucopoulos [242], pages 257–277.
317. J. Rothenberg. Prototyping as Modeling: What is Being Modeled? In H. G. Sol and K. M. van Hee, editors, *Dynamic Modelling of Information Systems*, pages 335–359. Elsevier Science Publishers B. V. (North-Holland), 1991.
318. K. S. Rubin and A. Goldberg. Object behavior analysis. *Communications of the ACM*, 35(9):48–62, September 1992.
319. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
320. H. Rybiński. On first-order-logic database. *ACM Transaction on Database Systems*, September 1987.
321. M. Saeki, H. Horai, and H. Enomoto. Software development process from natural language specification. In *Proceedings of the 11th International Conference on Software Engineering (ICSE'89)*, pages 64–73. IEEE Computer Society Press, 1989.
322. D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Model-theoretic fundations. Technical report, Laboratory for Foundation of Computer Science, Department of Computer Sceince, The Univ. of Edinburgh, 1992.
323. *Rett sats: Kartlegningskonferanser som utgangspunkt for bedriftsutvikling (In Norwegian)*. Senter for bedre arbeidsliv, 1991.
324. A-W. Scheer and A. Hars. Extending data modelling to cover the whole enterprise. *Communications of the ACM*, 35(9):166–171, September 1992.
325. D. Schuler and A. Namioka. *Participatory design: Principles and Practices*. Lawrence Erlbaum, 1993.
326. A. Schutz. *Collected Papers*. Njihoff, 1962.
327. J. R. Searle. *Speech Acts*. Cambridge University Press, 1969.
328. J. R. Searle. *Expression and Meaning*. Cambridge University Press, 1979.
329. J. R. Searle and D. Vanderveken. *Foundations of Illocutionary Logic*. Cambridge University Press, 1985.
330. A. H. Seltveit. An abstraction-based rule approach to large-scale information systems development. In Rolland et al. [315], pages 328–351.

331. A. H. Seltveit. *Complexity Reduction in Information Systems Modelling.* PhD thesis, IDT, NTH, Trondheim, Norway, 1994.
332. J. A. Senn. *Analysis & Design of Information Systems.* McGraw-Hill, 1989.
333. A. Sernadas, C. Sernadas, and H. D. Ehrich. Object-oriented specification of databases: An algebraic approach. In *Proceedings of the 13th VLDB*, pages 107–116, 1987.
334. C. Sernadas, J. Fiadeiro, and A. Sernadas. Modular construction of logic knowledge bases: An algebraic approach. *Information Systems*, 15(1):37–59, 1990.
335. C. Sernadas and J. Fiaderio. Towards object-oriented conceptual modeling. *Data & Knowledge Engineering*, 6(6), 1991.
336. C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication.* University of Illinois Press, 1949.
337. S. Shlaer and S. J. Mellor. *Object Lifecycles, Modeling the World in States.* Yourdon Press, 1991.
338. B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison Wesley, Reading, Massachusetts, 1992. 2nd edition.
339. Y. Shoham. Agent oriented programming: An overview of the framework and summary of recent research. In M. Masuch and L. Pólos, editors, *Knowledge Representation and Reasoning under Uncertainty: Logic at Work*, pages 123–129. Springer Verlag, 1994.
340. J. Siddiqi. Challenging universal truths of requirements engineering. *IEEE Software*, pages 18–19, March 1994.
341. H. A. Simon. The organization of complex systems. In Pattee [299].
342. G. Sindre. Abstraction of behavior network models. Technical report, IDT, NTH, Trondheim, Norway, 1987.
343. G. Sindre. RAPACITY - An approach to constructivity in conceptual modelling. Master's thesis, IDT, NTH, 1988.
344. G. Sindre. *HICONS: A General Diagrammatic Framework for Hierarchical Modelling.* PhD thesis, IDT, NTH, Trondheim, Norway, 1990. NTH report 1990:44, IDT report 1990:31.
345. G. Sindre and J. Krogstie. Process heuristics to achieve requirements specification of feasible quality. In K. Pohl and P. Peters, editors, *Second International Workshop on Requirements Engineering: Foundations for Software Quality (REFSQ'95)*, pages 92–103, Jyväskylä, Finland, June 12-13 1995.
346. J. Slonim. OO in the real world - success or latest fashion? In Müller and Georges [266], pages 440–441.
347. R. Snodgrass. Monitoring in a software development environment - a relational approach. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments, SIGPLAN, ACM SIGSOFT*, 1984.
348. A. Sølvberg. A contribution to the definition of concepts for expressing users' information systems requirements. In P. P. Chen, editor, *Entity-Relationship Approach to Systems Analysis and Design.* North-Holland, 1980.
349. A. Sølvberg, J. Krogstie, and A. H. Seltveit, editors. *Proceedings of the IFIP8.1 WC on Information Systems Development for Decentralized Organizations (ISDO'95)*, Trondheim, Norway, 21-23 August 1995. Chapman & Hall.
350. A. Sølvberg and C. H. Kung. *Information Systems Engineering.* Springer-Verlag, 1993.
351. P. G. Sorenson, J.-P. Tremblay, and A. J. McAllister. The metaview system for many specification environments. *IEEE Software*, 5(2):30–38, March 1988.
352. J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison Wesley, 1983.

353. S. Spaccapietra and C. Parent. View integration: A step forward in solving structural conflicts. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):258–274, April 1994.
354. R. Stamper. Semantics. In R. J. Boland Jr and R. A. Hirschheim, editors, *Critical Issues in Information Systems Research*, pages 43–78. John Wiley & Sons, 1987.
355. L. Suchman. *Plans and Situated Actions.* Canbridge University Press, New York, 1987.
356. A. G. Sutcliffe and N. A. M. Maiden. Bridging the requirements gap: Policies, goals and domains. In *Proceedings of the Seventh International Workshop on Software Specification and Design (IWSSD-7)*, pages 52–55, Redondo Beach, USA, December 6-7 1993.
357. E. B. Swanson and C. M. Beath. *Maintaining Information Systems in Organizations.* Wiley Series in Information Systems. John Wiley & Sons, 1989.
358. W. Swartout. GIST English generator. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-82)*, pages 404–409, Pittsburgh, USA, August 1982.
359. W. Swartout. The GIST behavior explainer. In *Proceedings of The National Conference on Artificial Intelligence*, pages 402–407, 1983.
360. W. R. Swartout. XPLAIN: A system for creating and explaining expert consulting programs. *Artificial Intelligence*, 21(3):285–325, March 1983.
361. K. D. Swenson, R. J. Maxwell, T. Matsymoto, B. Saghari, and I. Irwin. A business process environment supporting collaborative planning. *Journal of Collaborative Computing*, 1(1), 1994.
362. A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.
363. R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, January 1988.
364. Y. Tao and C. H. Kung. Formal Definition and Verification of Data Flow Diagrams. *The Journal of Systems and Software*, 1991.
365. R. D. Tavendale. A technique for prototyping directly from a specification. In *Proceedings of the 8th international conference on software engineering*, pages 224–229. IEEE Computer Society Press, 1985.
366. Tempora: Integrating database technology, rule-based systems and temporal reasoning for effective software. Technical Report ESPRIT Project 2469, Technical Annex, Tempora Consortium, October 17 1988.
367. Tempora final review. Technical report, Tempora Consortium, 1994.
368. A. H. M. Ter Hofstede, H. A. Proper, and T. P. Van der Weide. Exploiting fact verbalization in conceptual information modeling. *Information Systems*, 22(6/7):349–385, 1997.
369. I. Thomas and B. A. Nejmeh. Definitions of tool integration for environments. *IEEE Software*, 9(2):29–35, March 1992.
370. K. Thoresen, V. Keul, J. Bing, A. Pape, and T. C. Paper. *Omstilling med IT (In Norwegian).* NKS-forlaget, 1992.
371. C. Tomlinson and M. Scheevel. Concurrent programming. In W. Kim and F. H. Lochovsky, editors, *Object-oriented Concepts, Databases and Applications.* Addison-Wesley, 1989.
372. D. C. Tsichritzis and F. H. Lochovsky. *Data Models.* Prentice-Hall, 1982.
373. W. Twining and D. Miers. *How to do Things with Rules.* Weidenfeld and Nicholson, 1982.

374. F. van Assche, P. Layzell, P. Loucopoulos, and G. Speltincx. Information systems development: A rule-based approach. *Knowledge Based Systems*, 1(4):227–234, September 1988.
375. I. van Horebeek and J. Lewi. Are constructive formal specifications less abstract? *SIGPLAN Notices*, 25(5):60–68, 1990.
376. V. E. van Reijswoud and N. B. J. van der Rijst. Modelling business communication as a foundation for business process redesign: A case of production logistics. In Nunamaker and Sprague [278], pages 841–851.
377. V. van Swede and H. van Vliet. Consistent development: Results of a first empirical study of the relation between project scenario and success. In G. Wijers, S. Brinkkemper, and T. Wasserman, editors, *Proceedings of the 6th International Conference on Advanced Information Systems Engineering (CAiSE'94)*, pages 80–93, Utrecth, Netherlands, June 6–10 1994. Springer Verlag.
378. T. F. Verhoef and A. H. M. Hofstede. Feasibility of flexible information modelling support. In Iivari et al. [179], pages 168–185.
379. T. F. Verhoef, A. H. M. Hofstede, and G. M. Wijers. Structuring modelling knowledge for CASE shells. In Andersen et al. [8], pages 502–524.
380. R. Veryard and J. Dobson. Third order requirements engineering: Specification, change, and identity. In K. Pohl and P. Peters, editors, *REFSQ'95*, 1995.
381. I. Vessey and S. A. Conger. Requirements specification: Learning object, process, and data methodologies. *Communications of the ACM*, 37(5):102–113, May 1994.
382. I. Vessey and R. Weber. Some factors affecting programming repair maintenance. *Communications of the ACM*, 26(2):128–134, February 1983.
383. M. Vestli, I. Nordbø, and A. Sølvberg. Developing well-structured knowledge-based systems. In *Proceedings of the Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE'94)*, pages 366–373, Jurmala, Latvia, June 21-23 1994.
384. M. Vestli, I. Nordbø, and A. Sølvberg. Modeling control in rule-based systems. *IEEE Software*, pages 77–81, March 1994.
385. G. H. Von Wright. *An Essay in Deontic Logic and the General Theory of Action*. North-Holland, 1972.
386. R. Vonk. *Prototyping — The effective use of CASE technology*. Prentice Hall, 1990.
387. Y. Wand. An ontological foundation for information systems design theory. In *Proceedings IFIP*, Linz, Austria, 1988.
388. Y. Wand and R. Weber. On the ontological expressiveness of information systems analysis and design grammars. *Journal of Information Systems*, 3(4):217–237, 1993.
389. B. Wangler, R. Wohed, and S-E. Öhlund. Business modelling and rule capture in a CASE environment. In *Proceedings of the Fourth Workshop on The Next Generation of CASE Tools*, Twente, The Netherlands, 1993.
390. P. T. Ward. The transformation schema: An extension of the data flow diagram to represent control and timing. *IEEE Transactions on Software Engineering*, 12(2):198–210, February 1986.
391. A. I. Wasserman, P. A. Pircher, and D. T. Shewmake. Building reliable interactive information systems. *IEEE Transactions on Software Engineering*, 12(1):147–156, January 1986.
392. H. Weigand. Conceptual models in PROLOG. In T.B. Steel Jr. and R. Meersman, editors, *Database Semantics*, pages 59–69. Elsevier Science Publishers B.V., 1986.

393. R. Wieringa. Three roles of conceptual models in information system design and use. In E. Falkenberg and P. Lindgren, editors, *Information System Concepts: An In-Depth Analysis*, pages 31–51. North-Holland, 1989.
394. R. J. Wieringa. *Algebraic Fundations for Dynamic Conceptual Models.* PhD thesis, May 1990.
395. R. J. Wieringa, J-J. C. Meyer, and H. Weigand. Specifying dynamic and deontic integrity constraints. *Data and Knowledge Engineering*, 4:157–189, 1989.
396. G. Wilkie. *Object-Oriented Software Engineering - The Professional Developers's Guide.* Addison-Wesley, 1993.
397. H. Willars. *Handbok i ABC-metoden (In Swedish).* Plandata Strategi, 1988.
398. G. Willumsen. *Executable Conceptual Models in Information Systems Engineering.* PhD thesis, IDT, NTH, Trondheim, Norway, November 1993.
399. R. J. Wilson. *Introduction to Graph Theory.* Longman, New York, 3 edition, 1985.
400. T. Winograd and F. Flores. *Understanding Computers and Cognition.* Addison-Wesley, 1986.
401. R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, Englewood Cliffs, NJ, 1990.
402. M. Wirsing. *Algebraic Specification*, chapter 13, pages 677–780. Elsevier, 1990.
403. L. Wittgenstein. *Philosophical Investigations.* Blackwell, 1958.
404. M. Yang. *COMIS - A Conceptual Model for Information Systems.* PhD thesis, IDT, NTH, Trondheim, Norway, 1993.
405. M. Yang and A. Sølvberg. The new PPP: Its architecture and repository management. In *Proceedings of the Fifth Workshop on The Next Generation of CASE Tools*, Utrecht, Holland, 1994.
406. E. S. K. Yu. An organization modelling framework for multi-perspective information systems design. Technical Report DKBS-TR-93-2, Department of Computer Science, University of Toronto, Canada, 1993.
407. E. S. K. Yu and J. Mylopoulos. Using goals, rules, and methods to support reasoning in business process reengineering. In Nunamaker and Sprague [277], pages 234–243.
408. P. Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, 8(3):250–269, May 1982.
409. P. Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3):212–225, March 1991.
410. L. Zusne. *Visual Perception of Form.* Academic Press, 1970.

Index