A Prototyping Approach to Validation of Conceptual Models in Information Systems Engineering

Odd Ivar Lindland

Information Systems Group
Faculty of Electrical Engineering and Computer Science
The Norwegian Institute of Technology
Abstract

This thesis is motivated by the many reports that claim that the quality of an information system is determined early in the development process. Conceptual models are suitable vehicles for representing functional properties of a domain during problem formulation and requirements specification. Quality assurance of conceptual models are therefore important to establish a sound and stable foundation for the design and implementation of the information system. Verification and validation are central activities for quality assurance. Efficient tool support is important in order to improve the quality and productivity of the verification and validation processes.

Several factors determine the quality of a model. In this thesis we propose a taxonomy emphasizing the importance of identifying major quality goals and means/factors for achieving the goals. The taxonomy establishes relationships between a model and its language, its domain, and its users. Subsequently, three main quality types are identified, syntactic, semantic and pragmatic quality, addressing the three relationships, respectively.

In the taxonomy, expressiveness and visualization are central language properties for model validation. The lack of expressiveness of the employed languages in most contemporary CASE tools prevents easy model validation. Expressiveness and visualizability are influencing a model's comprehensibility which is a major influence for successful validation of a conceptual model. Despite the user-oriented nature of many conceptual models, they become increasingly difficult to read and fully understand when their size and complexity grow. Also, as the modeling languages have become more expressive and formal, system proposals becoming difficult to judge only by inspecting the models.

The approach which is particularly emphasized in this thesis is the prototyping approach. It is based on the assumption that it is difficult to judge the consequences of the model before it is realized in the final information system. The prototyping approach exploits the executability of conceptual modeling languages, and translate the associated models to an executable prototype. A prototype is a tangible software product which can be used to validate the dynamic properties of a model.

To show the feasibility of the prototyping approach a suitable systems development environment must be established. It is a major undertaking to establish an environment which is sufficiently wide-spectered. Much of my work therefore has been devoted to the design and implementation of an environment that meets the requirements to the above approach — the PPP (Phenomenon, Processes, and
Programs) environment. The PPP environment is running on Sun work-station under Unix and Sunview and has been developed using BIM-Prolog as the major implementation language. The details of the prototyping approach are shown by capitalizing on two translation assistants built within the PPP project: from PPP to Ada and from PPP to TEQUEL/C. We are using the assistants to explain and evaluate our approach.

The development of the PPP environment has shown the feasibility of using a translational prototyping approach in the context of graphical CASE environments. Modeling and verification support are major prerequisites for translating PPP models into executable code. One of the major achievements in this thesis has been to build a tool with such support facilities.

A major contribution of this thesis is an integration of a number of methodological components and tool components into one whole. The integrative work has been directed to improve the basis for quality assessment in IS engineering, in particular to improve model comprehensibility based on translational prototyping.
Preface

The work presented in this report is a thesis submitted to the Norwegian Institute of Technology for the doctor degree “doktor ingeniør”. The work has been carried out at the Information Systems Group, Faculty of Electrical Engineering and Computer Science, The Norwegian Institute of Technology, The University of Trondheim, Norway, from spring 1988 to spring 1993, under the supervision of Prof. Dr. Arne Sølvberg. The work was interrupted in 1989 due to military service.

Acknowledgements

I would like thank the following persons who, in addition to the author, have contributed to this thesis:

- Professor Arne Sølvberg for his continuous encouragement, guidance, and constructive criticism throughout the work. In particular, when I stumbled into blind alleys, his efforts in keeping my thesis on track have been invaluable in order to let me see the end of the work.

- Fellow students at the Information Systems Group: Rudolf Andersen, Jon Atle Gulla, John Krogstie, Andreas Lothe Opdahl, Anne Helga Seltveit, Guttorm Sindre, Vidar Vetland, Geir Willumsen, and Mingwei Yang for providing an inspiring working environment. In particular, Gulla, Krogstie, Sindre, and Willumsen, with whom I have been co-authoring several papers, and Yang with whom I have been developing the PPP software.

- All project and diploma students who have worked on different aspects of the PPP environment — teaching me its strengths and weaknesses.

- Finally, I would thank my dear wife Hanne for being patient with me (almost) all the way through.

Trondheim, May 1993,

Odd Ivar Lindland
#Contents

1 **Introduction**
   1.1 About the Problem ................................................. 1
   1.2 About the Work ................................................... 3
       1.2.1 The “Theoretical” Part ..................................... 4
       1.2.2 The “Practical” Part ....................................... 6
   1.3 Major Contributions ............................................. 7
   1.4 Related Work ..................................................... 7
       1.4.1 Related Doctoral Thesis ............................... 8
   1.5 Organization of the Thesis .................................... 9

2 **Terminology and Motivation** ........................................ 11
   2.1 Information Systems and Software ............................. 11
       2.1.1 Information Systems ...................................... 11
       2.1.2 Software .................................................... 13
   2.2 The “Wickedness” of IS Engineering .......................... 13
   2.3 Quality Assurance in IS Engineering .......................... 14
       2.3.1 Verification and Validation ............................ 15
       2.3.2 Tool Support for Quality Assurance ..................... 16
       2.3.3 Quality Metrics .......................................... 17
       2.3.4 The Need for Quality Assurance of Requirements ..... 17
   2.4 Information Systems Requirements ............................ 18
       2.4.1 Domain Description and Requirements Specification .... 18
       2.4.2 Feasibility Study .......................................... 19
       2.4.3 Functional and Non-Functional Requirements ............. 20
       2.4.4 Requirements Engineering ............................... 20
   2.5 Conceptual modeling ............................................. 20
       2.5.1 Major Purpose ............................................. 21
       2.5.2 Knowledge Capture ........................................ 21
       2.5.3 Model Construction and Knowledge Representation ..... 22
   2.6 Towards Model Quality Assurance .............................. 22

3 **A Taxonomy for Model Quality** ...................................... 25
   3.1 Existing Taxonomies ............................................ 25
       3.1.1 Kung’s Taxonomy ........................................... 26
       3.1.2 Roman’s Taxonomy ......................................... 27
       3.1.3 Yeh’s and Davis’ Taxonomies ............................ 27
       3.1.4 The Need for an Improved Taxonomy ..................... 28
   3.2 Basic Terminology of the Taxonomy ............................ 28
3.3 Syntactic Quality ............................................. 31
3.4 Semantic Quality .......................................... 32
3.5 Pragmatic Quality ......................................... 35
3.6 Using the Taxonomy .................................. 37

4 Conceptual Modeling Languages and CASE tools 41
4.1 Conceptual Modeling Languages ..................... 41
  4.1.1 Data-Oriented Languages .......................... 42
  4.1.2 Process-Oriented Languages ..................... 44
  4.1.3 Behavior-Oriented Languages ..................... 44
  4.1.4 Other Language Perspectives and Integrated Languages .. 46
4.2 CASE tools .................................................. 48
  4.2.1 The First Generation CASE Tools ................. 48
  4.2.2 The Second Generation CASE Tools ................. 48
4.3 Towards Model Quality Assurance in CASE Tools ................. 50

5 Three Validation Approaches 51
5.1 The Complexity Reduction Approach .................. 51
5.2 The Presentation Approach ............................... 53
5.3 Towards Combined Approaches .......................... 54
5.4 The Prototyping Approach — Baseline ................. 55
  5.4.1 Prototyping ............................................. 55
  5.4.2 Prototyping — A Taxonomy .......................... 57
  5.4.3 Towards Translational Prototyping ................. 58
  5.4.4 Translations ............................................. 59
  5.4.5 A Compiler View of the Translations ................. 61
  5.4.6 Towards a Tool Supported Prototyping Approach ................. 63

6 Tool Support for the Prototyping Approach 67
6.1 Requirements to the Modeling Environment ............. 68
6.2 Requirements to the Repository .......................... 70
6.3 Requirements to the Translation Assistant ............. 71
6.4 Requirements to the Prototyping Environment .......... 72
6.5 A Short Survey of Related Work .......................... 75

7 The PPP Environment — Realizing the Prototyping Approach 77
7.1 The PPP Modeling Approach ............................... 77
7.2 Domain Description of a “Simple” Bank ................. 79
7.3 Some Remarks on the Evaluation .......................... 80

8 The PPP Language 81
8.1 The PhM Language ............................................. 81
8.2 The PrM Language ............................................. 82
8.3 The PLD Language ............................................. 85
8.4 The UID Language ............................................. 87
8.5 The Integrated PPP Language ............................. 87
8.6 Properties of the PPP Language/Model .................. 89
  8.6.1 Syntax .................................................. 89
11.2.2 Towards Evolutionary Prototyping .......................... 135
11.2.3 Towards an Integrated Prototyping Environment .......... 135
11.2.4 Towards an Integrated Validation Environment ........... 135
11.2.5 Integrating Configuration Management ...................... 136
11.2.6 Comparative Experimental Surveys .......................... 137
Chapter 1

Introduction

This thesis is about assessing the quality of conceptual models that are developed through the application of CASE tools. There are several approaches to quality assessment. The most important are verification, validation, the use of standards, metrics, project management routines, and configuration management techniques. In this thesis we concentrate on validation techniques. There are different approaches for validation: prototyping, complexity reduction, and presentation. The prototyping approach is emphasized in this thesis. There are several ways of providing a prototyping approach: translation, interpretation, and use of fourth generation language. We have exploited the use of translations.

Figure 1.1 shows the thesis' context. It illustrates the above dependencies, while emphasizing both the goals and the means/factors for meeting the goals. The main "components" of the thesis are boldly depicted. In this chapter we will discuss the different components and their relationships. They form the background for the problem formulation and objectives of the thesis that are described in Section 1.1. Also, the figure shows the approach to the problem that is taken by this thesis. This is further explained in Section 1.2. The main contributions of the thesis are listed in Section 1.3, whereas related work is discussed in Section 1.4. The organization of the thesis is described in Section 1.5.

1.1 About the Problem

The concept of quality of information systems is intangible, subjective, and complex. A wide range of important features have been defined by various authors. They emphasize different quality attributes like reliability, dependability, and user-friendliness. Quality problems of information systems have mostly been addressed in the later phases of the development process. Particularly, a wide range of verification and validation techniques like debugging, testing, and program verification
have been proposed.

However, experiences have shown that the quality of an information system is largely determined early in the development cycle, i.e. during problem formulation and requirements specification. Moreover, errors introduced at this stage are usually much more expensive to correct than errors introduced during design or coding. Thus, it would be appropriate to prevent, detect, and correct errors as early as possible in the development process.

With the increased emphasis on user participation in development projects, conceptual models have emerged as suitable vehicles for representing functional properties of a problem domain early in the development process. Conceptual models are assumed to be close to the way humans perceive the domain, and should encourage active involvement by users. They also support the system developers in investigating the problem domain and users’ information needs, as well as form a basis for the subsequent design and implementation of the information system. Moreover, they provide a documentation of the system from the user’s point of view and set a standard against which the design and implementation are tested.
1.2. About the Work

Verification and validation are central activities to establish an error-free model and hence improve the model quality. Verification aims at removing syntax errors and locational inconsistencies from the model, whereas validation emphasizes on letting the model express relevant user requirements and needs. Validation requires active user participation.

The quality assessment potential of a model depends on several factors. In this thesis we will emphasize the following:

- *Properties of the conceptual modeling language.* Because of the wide spectrum of errors that may exist, different and often contradictory properties are desired of the language in order to be able to construct a model free of errors. On one hand, for verification purposes we would want a language which has the precision of mathematics; on the other hand, for validation purposes we need the understandability and accessibility of natural language.

- *Tool support.* Verification and validation are time consuming and error sensitive when done as manual activities. Computerized assistance providing support for verification and validation will be important for improving the quality and productivity of the quality assurance process.

So, based on this brief description of the problem area, the following objective of the thesis can be stated:

*We will investigate which properties that a conceptual modeling language and its CASE tool must have in order to support quality assurance of a conceptual model. The approach shall be evaluated by trying it out in a CASE environment which can support the chosen modeling languages.*

In the sequel we will explain how the above objective has been approached and motivate for the main decisions made during work on the thesis.

1.2 About the Work

The emphasis in this thesis will be to analyze and experiment with different technical factors that may affect the quality of conceptual models. The above objective is too wide-spectered to be fully solved within the scope of one single thesis. So, the work is generally characterized as follows:

- Identify goals and means for achieving conceptual models of satisfactory quality.
• Establish a particular approach to model quality assurance, that can be subjected to experimentation.

• Capitalize on previous and ongoing works in the Information Systems Group.

We distinguish between the “theoretical”/analytical part and the “practical”/experimental part of the thesis.

1.2.1 The “Theoretical” Part

The theoretical work has concentrated on verification and validation of conceptual models. We have proposed a taxonomy for model quality. The taxonomy establishes relationships between model, language, domain, and user. Three main quality types are identified: syntactic, semantic, and pragmatic quality. Quality goals and means for achieving the goals are discussed for each quality type.

In the taxonomy, expressiveness and visualizability are central language properties for obtaining easy quality assessment of information systems models. Although the modeling languages supported by contemporary CASE tools are visual, most of them have limited expressiveness. The limited support for model quality assurance in most contemporary CASE tools stems from an insufficient language basis.

Verification and validation serve different purposes. The focus of this thesis will be on validation of conceptual models. Since validation requires that the model is discussed among users and developers, a major property of the modeling language and the associated models is comprehensibility. By their visual nature, many conceptual modeling languages provide this property. However, the models become increasingly difficult to read and to fully understand as their size and complexity grow. Also, as modeling languages have become more expressive and formal, models are difficult to judge only from model inspection. Consequently, the comprehensibility of the models suffer, and should be improved by validation techniques.

In collaboration with fellow doctoral students Gulla, Willumsen, and Seltveit, a distinction between three main approaches for improving model comprehensibility have been established [154]. First, the complexity reducing approach addresses the fact that when conceptual models are large and complex, the users may get confused by all the details included in the model. By reducing the complexity of the model, it is envisaged that the comprehensibility of the model will be improved, and thus make the validation process more feasible. Our approach deals with this problem either by grouping detailed specifications, focusing on relevant views of the model, or by improving the layout of the model. Second, many conceptual models are difficult to read either because of their formality or because of the readers unfamiliarity with the modeling language. The syntax and the semantics of
formal modeling languages may easily confuse unskilled users. The modeling concepts of the languages may differ from what the user is already acquainted with. Various presentation techniques try to alleviate this by presenting a model in some other language with which the user is already familiar.

The approach which is particularly addressed in this thesis is the prototyping approach. The architecture of the approach is shown in Figure 1.2 and is described by the following major step: modeling, translation, execution, and validation. Firstly, the model is constructed by means of a conceptual modeling language. During the construction, the model is verified in order to remove syntactical errors and inconsistencies. Secondly, when the model has reached sufficient size and quality, the executable properties of the model may be exploited by translating it into an executable prototype. Thirdly, by executing the prototype, the behavior of the model can be revealed. Finally, the behavior is validated with respect to the expected or intended behavior in the domain. The model is modified based on invalidities in the model. Consequently, the approach is cyclic and terminates when the model is sufficiently stable.

The requirements to the realization of the prototyping approach within the context of a CASE tool are discussed. For discussing the details of a translation assistant, we have used a traditional compiler architecture. Furthermore, the environment for executing the prototype — the prototyping environment — is crucial for the model validation. Requirements to such an environment are established as well.
1.2.2 The "Practical" Part

To show the feasibility of the prototyping approach a suitable systems development environment must be established. It is a major undertaking to establish an environment which is sufficiently wide-spectered. When the work with this thesis started there were no languages and CASE tools available in the market that satisfied our needs. Much of my work therefore has been devoted to the design and implementation of an environment that meet the requirements to the above approach — the **PPP (Phenomenon, Processes, and Programs) environment**. The environment has been developed as a part of this thesis, in collaboration with other members of the Information Systems Group. The PPP environment is running on Sun work-stations under Unix and Sunview and has been developed using BIM-Prolog as the major implementation language. Modeling and verification support are major prerequisites for translating PPP models into executable code. One of the major achievements of this thesis has been to build a tool with such support facilities.

PPP provides modeling support for a set of four integrated languages, which cover different aspects and stages of systems development, and are based on well established languages. The **PrM (Process Model)** language is used to describe dynamic aspects. It is based on the traditional DFD, with added constructs for better precision and increased expressiveness. The **PhM (Phenomenon Model)** language is an extension of the entity-relationship model, and includes many features of newer semantic data models [114]. The **PLD (Process Life Description)** language is used to specify process logic of bottom-level processes. It has many similarities with block-structured, program design languages [144]. The **UID (User Interface Description)** language [74] is used to describe static and dynamic aspects of user interfaces, based on the possibilities offered by graphical user interface technology.

Compared to most contemporary CASE languages, the PPP language has better expressiveness. Moreover, PrM and PLD have executable properties. Particularly, we have exploited the properties of these sub-languages to automatically translate PPP models into executable prototypes so that the dynamic properties of the model can be validated.

Within the PPP project, several translation assistants have been developed. In this thesis we capitalize on the following two translation assistants:

1. from PPP models to Ada and
2. from PPP models to TEQUEL/C

We use the translation assistants to explain the details of the prototyping approach. The approach is also evaluated in the validation context established in the theoretical part of the thesis.
1.3 Major Contributions

A major contribution of this thesis is an integration of a number of methodological components and tool components into one whole. The integrative work has been directed to improve the basis for quality assessment in IS engineering, in particular to improve model comprehensibility based on translational prototyping.

During this work it has been natural to incorporate results from earlier and parallel work by other members of the Information Systems Group. Several relevant venues have also been explored by formulating proper tasks for diploma students that I have supervised at the Institute.

My own contribution are in particular related to the following:

- Integrating methodological approaches and tool components into one whole. Subsequently, terminology clarifications have been carried out. This is particularly manifested in the taxonomy for model quality.

- Establishing and discussing the mutual effects of integrating the different components/approaches.

- Realizing major parts of the PPP environment, in particular its modeling environment.

The major contribution of the thesis as whole may be summarized as follows:

- The PPP environment has shown the feasibility of using a translational prototyping approach in the context of graphical CASE environments. Since we have built our approach on well established modeling languages, we are laying the ground for a future technology transfer into contemporary practice.

- Moreover, the thesis has, apart from proposing and experimenting with a particular approach for validating conceptual model, contributed to the understanding of quality assurance of conceptual models in general.

- Finally, the work has laid the ground for analyzing and experimenting with other validation approaches and different combinations of them as well. Ongoing works in this area are outlined below.

1.4 Related Work

This thesis integrates several methodological approaches and tool components that are known from the literature. Comprehensive discussions of various aspects of each component have been carried out by different authors. Eloquent
discussions on prototyping are provided by Vonk in [149] and by Blum in [14], whereas Balzer thoroughly elaborates on the use of translations in the development process in [9, 10]. The potential of executable properties combined with the application of visual languages for systems development is promoted by Harel [64, 65]. Harel is also the originator of Statcharts ([63]) which is the major modeling language in STATEMATE ([66]) — a sophisticated commercial CASE tool supporting the development of real-time systems. Harel’s discussion in [65] has been the main inspiration source for establishing the requirements to the prototyping environment in Section 6.4.

Although different components/approaches are emphasized by different authors, explicit discussions of their relationships and mutual benefits are rarely found. An exception is Agresti, who in [2] discusses the integration of prototyping, translations, and executable specifications. This thesis has further elaborated on different aspects of the integration, and discusses the approaches/components within the realm of model quality assurance.

1.4.1 Related Doctoral Thesis

This thesis has been developed in relation with the following forthcoming thesis by PhD-students of the Information Systems Group:

Mingwei Yang ([158]) has developed major parts of the PPP environment in cooperation with me. Among other things, Yang is in his thesis extending the verification support of PrM models by using canonical ports.

Geir Willumsen ([154]) is developing an execution module which can interpret PPP models, and generate an execution trace with query facilities. His work is linked to the work of Gulla.

Jon Atle Gulla ([57]) is developing a general explanation module that can explain behavior of PPP models in response to questions posed by users or developers. As such, input from Willumsen’s execution trace is used.

Anne Helga Seltveit ([126]) is proposing a general framework for reducing complexity in large models. The framework is applied on large and complex PPP models.

Rudolf Andersen ([4]) addresses the problems of team cooperation in distributed development environments. Particularly, an architecture for configuration management of PPP models is established.
1.5 Organization of the Thesis

The organization of the thesis is as follows:

- This chapter (1) has given an overall view of the problem and of the work that has been carried out in order to solve the problem.

- In *Chapter 2* a thorough motivation for the thesis is given along with an introduction of the basic terms used in this thesis.

- A general taxonomy for model quality is proposed in *Chapter 3*.

- In *Chapter 4* a brief survey of trends in the development of conceptual modeling languages and CASE tools given.

- *Chapter 5* presents three approaches for improving model comprehensibility. The baseline for the prototyping approach is established.

- The requirements to a tool for supporting the prototyping approach is presented in *Chapter 6*. Requirements to the translation assistant using a "compiler view" are discussed. The features of the prototyping environment are described as well.

- An introduction to the PPP environment is given in *Chapter 7*. A description of an example domain of banking routines is included.

- *Chapter 8* explains the PPP language by modeling examples from the bank domain.

- The PPP tool is presented in *Chapter 9*. The architecture and implementation platform are described. Moreover, the modeling and verification support are explained along with a modeling session.

- The validation support using the translational prototyping approach is presented in *Chapter 10*. Two translation assistants are explained and their potential for validating a PPP model is discussed.

- Finally, *Chapter 11* concludes the thesis. The achievements are outlined and directions for further work are established.
Chapter 2

Terminology and Motivation

The previous chapter introduced the problem area addressed in this thesis and outlined the major characteristics of our work to the problem. This chapter will more thoroughly motivate for tool supported quality assurance of conceptual models. Also, a further explanation of some central terms used in the thesis is given.

In Section 2.1 the basic notions of information systems and software are explained. The "wickedness" in information systems engineering is introduced in Section 2.2. Section 2.3 presents activities in the quality assurance process. A motivation for addressing these issues early in the development process is given. Section 2.4 explains central terms in connection with information system requirements. These are related to conceptual modeling in Section 2.5. Finally, Section 2.6 bridges the discussion towards quality assurance of conceptual models.

2.1 Information Systems and Software

2.1.1 Information Systems

Information systems provide support for reaching organizational goals in a timely and cost-beneficial manner. They have the following properties [6]:

- they provide a historical record of the activities in the organization;
- they provide means to meet the internal and external reporting requirements of the organization;
- they provide the info/infra-structure — a system of data and information flow and a structure of responsibility within the organization;
they provide direct support for the operational activities of the organization;

they provide essential information to the decision-making process of the organization.

Since computerized systems were introduced in the mid-fifties to support well-defined, though limited, clerical tasks, there have been dramatic changes in: (1) their use, (2) how users interact with them, (3) the role they play in organizations, and (4) the way they are developed [25]. From being batch-oriented, "long-life" systems with a centralized architecture, information systems now play an essential role in most organizations and support a wide range of organizational activities. Moreover, the architecture is distributed through networks and the interaction with the systems is frequent and versatile.

Information systems are usually seen as systems for support of human organizations. They are therefore often distinguished from other systems like [116]:

**Real-time systems** which measure/analyze/control events in real time. A real-time system collects data from its external environments, and translates the information as required by the application. Furthermore, the system responds to the environment by a monitoring component which coordinates the other system components so that real-time response can be maintained. If such a requirement cannot be met, it can lead to disastrous results. Typical examples of real-time systems are found in e.g, missile controls and autopilots of air-planes.

**Embedded systems** have many similar characteristics of real-time systems, but have a more limited functionality. Generally, they are embedded in a wide range of electronic equipments such as video recorders, digital watches, micro-wave ovens, etc. Often, user interaction is restricted to simple keypads and displays.

**Scientific systems** or technical systems are oriented towards problems that can be solved by "number-crunching" algorithms. Thus, heavy computational power is required for such systems. Astronomy, stress analysis, molecular biology are some application areas for this type of systems.

**Expert Systems** have been introduced along with the expanding area of artificial intelligence. They make use of nonnumerical algorithms and heuristics to solve complex problems. Many systems are based on reasoning around a knowledge (facts and rules) base using different inference mechanisms. Typical application areas have been pattern recognition, theorem proving, and game playing.

Although these system classes are different from information systems, the borders between the system classes are vague. As the versatility and complexity of their problem domains grow, future systems are envisaged to integrate systems components from different system classes.
2.1.2 Software

A fundamental component of all system classes is software. Giddings distinguishes between two major types of software: domain independent and domain dependent [51]. Moreover, the latter type is either experimental or embedded\footnote{These three software types corresponds to Lehman’s S-Programs, P-Programs, and E-programs, respectively [82].}. The three types are explained as follows [51]:

**Domain Independent Software** is characterized by its general applicability. Typical examples are numerical algorithms, word processing systems, etc. This type of software is developed under a contract which is predefined and fixed.

**Domain Dependent Experimental Software** is characterized by an intrinsic uncertainty about the problem domain. A major goal of the development process is therefore to search for knowledge about the domain and to produce software which are useful for testing hypothesis or exploring unknown characteristics of the domain. For instance, domain models from areas like economy and physics can be used as hypothesis. The purpose of the software is to gather and analyze data (observations) from the domain in order to reject, modify, or accept the models.

**Domain Dependent Embedded Software** is characterized by interdependence between the problem domain and the software. The use of the software may change the nature of the problem domain. Consequently, the requirements to software will change, forcing further tailoring of the software.

In this text, we are mainly concerned with information systems which contain domain dependent software. The software is developed and tailored to specific needs and requirements found in the domain. Despite this focus, the following discussion is not restricted to development of domain dependent information systems only.

2.2 The “Wickedness” of IS Engineering

It is generally expected that information systems embedded in organizations will increase the competitive advantage of the organization. However, the process of developing information systems, *Information Systems Engineering (IS Engineering)*, is not free of problems. Some of the problems — commonly referred to as the software crisis — manifest themselves in overrunning budgets of development projects. Moreover, the systems often does not meet the user’s requirements.
The underlying reasons for the problems are highly complex and have both organizational, economic, and technical aspects [69, 99]. Some surveys have indicated that over-optimistic planning, underestimation of the scope, and poor contracting practices are major reasons for overrunning budgets [140, 146].

Other authors emphasize that a main reason for project failure stems from the nature of information systems per se and that IS engineering is fundamentally different from and more complex than other engineering disciplines. Among others, Brooks [22] claims that a central cause of the difficulties is that software is abstract and not materialized. Moreover, software is difficult to visualize and corresponds to complex, non-specific set of human behaviors. Moreover, information systems usually have a larger state space than other systems and that there is no repetition of components. Rittel states that development of domain dependent software is inherently “wicked” and may be characterized in the following way [118]:

- “Wicked problems” have no definitive formulation,
- there are no stopping rules for solving “wicked problems”,
- solutions to “wicked problems” are neither correct nor false,
- every “wicked problem” can be considered as a symptom of another problem, and
- every “wicked problem” is essentially unique.

The “wickedness” of IS Engineering implies that information systems never can be right or wrong, only more or less good. The quality of the information system must therefore to a large extent be decided by the engineering practice. Consequently, quality assurance is an important activity.

### 2.3 Quality Assurance in IS Engineering

Quality assurance is a well established area in a wide range of engineering fields. In IS Engineering, quality assurance has traditionally been coupled to software quality. According to the *IEEE Standard Glossary of Software Engineering Terminology*, ([41]), quality assurance is defined as:

> A planned and systematic pattern of all actions necessary to provide adequate confidence that a product (here software) conforms to established technical requirements.

In the following we will briefly introduce some topics which are central in software quality assurance.
2.3. Quality Assurance in IS Engineering

![Diagram of verification and validation of software](image)

Figure 2.1: Verification and validation of software (from [82]).

2.3.1 Verification and Validation

*Verification* and *Validation* are two basic activities for obtaining software quality. A well-known distinction made by Boehm is that verification and validation should answer the questions "*Am I building the system right?*" and "*Am I building the right system?*", respectively [17]. These phrases correspond well with the definitions given in [141]:

Verification is the process of determining whether the products of a given development phase satisfy the requirements established during the previous phase.

Validation aims at determining the correctness of the final program or software produced from a development project with respect to user's needs and requirements.

In Figure 2.1 we have illustrated the distinction. The object for verification and validation is a program. Whereas verification aims at checking the program's correctness relative to its specification, validation emphasizes comparing the program's output with relevant properties of the problem domain. Often, the output is judged by user groups. Thus, validation requires some sort of user involvement. If the program fails to meet the specification or if its outputs do not correspond user's intention, both verification and validation may impose changes on the program, on the specification, or on the user requirements.
Three different well-known verification and validation techniques are program testing, code inspection, and program verification [133]:

**Program Testing** is done by executing the program in order to find logical errors and to determine its reliability. Because of the enormous number of logical paths even in small computer programs, exhaustive and complete testing is impossible. The generation of "good test cases" along with the selection of appropriate testing strategies is vital to the success of program testing.

**Code inspection** is a human's inspection of software. It involves that the programmer allows an inspection team to read his code. The team has studied the specification and design of the code prior to the inspection. During the inspection itself, the programmer explains to inspection team how he has implemented the design and the team responds if anomalies and errors are detected in the code.

Whereas testing requires that the detected error is corrected prior to further testing, several errors can be detected in a single inspection session. The time of correcting all errors at once is much less than that required to detect and correct them one by one. This is the main advantage of code inspection as compared to testing.

**Program Verification** uses mathematical methods to check the internal correctness of the program, and the consistency between the program and its specification. Program verification has only proved feasible on small programs. The technique is still mainly in the research realm. A major weakness of program verification is, according to Sommerville, its inherent assumption about a correct program environment. In fact, program verification cannot detect environmental effects, such as the case with testing [133].

A further discussion of these techniques is found in [133], whereas overviews of different verification and validation techniques for software and other "deliverables" during the development process are given in [18, 150].

### 2.3.2 Tool Support for Quality Assurance

Verification and validation are time consuming and error sensitive when performed as manual tasks. Tool support is therefore important in order to improve productivity and quality of such tasks. Along with the techniques above, a wide range of support tools have been developed [133]. For instance, syntax directed editors force the programmer to write software which is free of syntactical errors. Program debuggers and test data generators support program testing.

As shown in Figure 2.1, verification and validation activities may create changes in the software. By modifying the software, new errors may be introduced and further verification and validation are required. To properly manage the iterative nature of the change process, support for configuration management is important.
2.3.3 Quality Metrics

Another important area of software quality assurance is the establishment of quality metrics which [41] defines as:

A quantitative measure of the degree to which software possesses a given attribute that affects quality.

Work on quality metrics aims at establishing relationships between the different quality attributes and express the relationships in the form of mathematical equations.

![Diagram of McCall's taxonomy for software quality](image)

Figure 2.2: Parts of McCall's taxonomy for software quality (adapted from [33]).

Generally, formal equations are derived from statistical and experimental analysis on how different criteria affect factors like usability, efficiency, reliability, maintainability, etc. In Figure 2.2 parts of McCall's taxonomy for software quality is shown [33]. The relationships between different criteria and factors are indicated. A thorough treatment of quality metrics is found in [45].

2.3.4 The Need for Quality Assurance of Requirements

Although quality assurance is an important activity to obtain confidence in the software, errors detected at this stage in the development process are expensive
compared to earlier detection and correction. Davis presents 5 "laws" which emphasize the importance of avoiding errors as early as possible in the development process [36]. The laws are stated in Figure 2.3. References to empirical surveys supporting each law are included.

1. The later in development life cycle that software a error is detected, the more expensive it will be to repair [16, 34, 43].
2. Many errors remain latent and are not detected until well after the stage at which they are made [20].
3. There are many requirements errors being made [27, 137].
4. Errors made in requirements specifications are typically incorrect facts, omissions, inconsistencies, and ambiguities [13].
5. Requirement errors can be detected [13, 24].

Figure 2.3: Davis’ 5 “laws” on requirement errors (from [36]).

Yeh summarizes the consequence of the laws as follows (quoted from [160]):

"...In short, because the requirement phase comes so early in the development, it has a tremendous impact on the quality (or lack thereof) of the development effort and the final product..."

We will now direct the attention to the early stages of the development process that are concentrated around information systems requirements.

### 2.4 Information Systems Requirements

#### 2.4.1 Domain Description and Requirements Specification

Information systems requirements — often called user requirements — are based on expressions of user needs and hopes about a future system that may improve the existing manual routines or outdated computerized system. Usually, users' needs are extended with information about the problem to be solved. Thus, relevant information about the domain are expressed in a (problem) domain description.

Information systems requirements are usually expressed in a document that is often called the requirement specification. According to Yeh, the requirement specification should contain (quoted from [159]):
"a set of precisely stated properties or constraints which a software system must satisfy."

This definition implies that requirements should be testable. As such, they should be distinguished from system goals which express general characteristics that the system should exhibit [133].

Figure 2.4: The requirement specification as a constrained solution space (adapted from [36]).

Several authors regard a requirements specification as a document which constrains the solution space to the problem at hand [36, 132, 160]. This view is illustrated in Figure 2.4. The outer rectangle represents the set of all possible solutions to the problem. Requirements (R1 — R4) constrain the solution space. An acceptable solution should meet all requirements. Solutions S1, S3, and S4 that are in the shaded area will be unacceptable since at least one requirement is not met, whereas solutions S2, S5, and S6 in the white area of the solution space are acceptable.

### 2.4.2 Feasibility Study

A feasibility study can be carried out as part of building the requirements specification and supplement the above view of the solution space. The study aims at finding the most acceptable solution or whether there exists an acceptable solution at all. Several alternative solutions are evaluated — often guided by a cost-benefit analysis. Thus, the estimated costs of the requirements are weighed against the projected benefits of the corresponding system. Typical benefits of developing an information system may be improved organizational productivity, reduced labor costs, etc. Such benefits have to be weighed against typical cost factors like development and maintenance costs, training costs, etc.
Ideally, to minimize the risk of choosing unacceptable solutions, a cost-benefit analysis should be carried out several times throughout the project [19].

2.4.3 Functional and Non-Functional Requirements

Information systems requirements are generally divided into [122, 160]:

- *Functional requirements* that are related to the business functions and the information routines that are subject to computerization. Furthermore, they focus on the interaction between the systems components and its environment. So, a large part of the functional requirements can be directly derived from a domain description.

- *Non-functional requirements* that address the effectiveness and timeliness of the information systems. Typical non-functional requirement are related to aspects like performance, reliability, security, economy, performance, portability, userfriendliness, etc. Discussions of non-functional requirements are found in [122, 160].

As will be motivated later, we will in this text concentrate on functional requirements.

2.4.4 Requirements Engineering

*Requirements engineering* has emerged as a research field that aims at managing the process of producing requirements for a software system. Often, the process is split into two major activities [26, 36, 61]. The first activity emphasizes on eliciting requirements from the problem domain and representing them in a requirement specification, whereas the second activity aims at analyzing the quality of the specification. Both activities will be further discussed in the sequel, but in the context of conceptual modeling.

2.5 Conceptual modeling

Conceptual modeling — the overall process of building a conceptual model — has strong relationships with requirement engineering. These relationships and other characteristics of conceptual modeling are described in the sequel.
2.5. Conceptual modeling

2.5.1 Major purpose

The purpose of conceptual modeling is to capture and document the various structures, functions, events, etc., in the problem domain that are relevant for the development of the information system. The conceptual modeling process should apply terms that users in the problem domain are familiar with. The conceptual model is expected to represent the relevant domain knowledge in a user-oriented manner. Thus, a conceptual model is a result of how human perceives the problem domain [80]. Moreover, the conceptual model should express the system’s behavior and structure and the system’s interaction with its environment.

The roles of a conceptual model have been stated by many researchers as follows [80, 132, 160]:

- It serves as a model of reality which gives insight into the problem domain. In other words, its construction enables the system analysts to reach a better understanding of the domain and the users needs.

- It serves as a common reference framework which is used during the system analysis phase to communicate within the development team and with future users of the system.

- It serves as a basis upon which the design and implementation of the information can be carried out and against which the design can be tested.

- It serves as a part of the documentation to be used during the maintenance phase to facilitate modifications and enhancements of the system.

Based on these roles, a conceptual model can be regarded as a bridge between the domain description and the requirements specification. In fact, a conceptual model will implicitly represent functional requirements to the future information system.

Furthermore, a conceptual model represents domain knowledge. Thus, conceptual modeling can be regarded as a bridge between IS engineering and knowledge engineering [93]. This view is further elaborated on throughout the thesis.

Below, we will discuss the following major activities/characteristic of conceptual modeling: knowledge capture and model construction.

2.5.2 Knowledge Capture

A main activity in the conceptual modeling process is to capture relevant knowledge that can be used as a basis for the functional requirements to the information
system. Traditionally, this activity has involved document collections/inspections, questionnaires, and interviews.

In knowledge engineering, however, knowledge acquisition techniques have been introduced for capturing expert knowledge and domain knowledge [67]. For instance, observation of the behavior of a domain expert is an alternative to interviewing the expert. Observation avoids potential biases introduced by the interviews. A related technique is protocol analysis which extracts knowledge from a record (or protocol) of events and statements made during a domain expert's performance. Audio-visual equipments are suitable tools for this technique. Furthermore, the acquisition part can be guided by structuring techniques such as card sorting and repertory grids.

Although some of these techniques may supplement the traditional techniques above and be used within an IS engineering context [42, 52], this methodological area is not addressed in this thesis.

2.5.3 Model Construction and Knowledge Representation

In order to represent the captured knowledge and hence constructing the conceptual model, a conceptual modeling language is used. A suitable language should contain a set of modeling constructs that symbolize relevant structural and behavioral properties of the problem domain [63, 83]. A model is constructed by "instantiating" modeling constructs with terms and concepts that domain users are familiar with.

From a knowledge engineering viewpoint, a conceptual modeling language can be regarded as a way of representing knowledge and thus supplement traditional representation techniques like logics, frames, semantic networks, and production rules [127].

2.6 Towards Model Quality Assurance

As with software quality, verification and validation are central activities to assure model quality. Although the definitions given in Section 2.3.1 applies well to verification and validation of conceptual models, some small modifications are necessary. These are shown in Figure 2.5. Here, the conceptual model is the verification and validation object. Verification aims at checking the internal quality of the model. Validation, on the other hand, aims at checking whether the model corresponds to relevant domain knowledge. In contrast to verification, validation requires some sort of user involvement. Although verification and validation serve different purposes, there are strong dependencies between them. In fact, many validation techniques require that the model has been verified in advance. This will be further discussed throughout the thesis.
As indicated in Figure 2.5, both verification and validation may impose changes on the model or on the way the problem domain is viewed. Changes may create new errors in the model, which requires further verification and validation. Quality assurance of the model may also trigger further knowledge capture and model construction. Thus, conceptual modeling in general and model quality assurance in particular are cyclic processes. As stated in Section 2.3.2, configuration management is important in order to properly control these processes. Within the scope of this text, however, configuration management will not be emphasized.

Conceptual modeling in general and model quality assurance in particular are time consuming and error sensitive activities. To ease the developer’s burden and to ensure both quality and productivity of the modeling processes, tool support is important. Tool support will become increasingly important as the size and complexity of a model grows. Tool support for verification and validation of conceptual model should be provided within the framework of the emerging technology of computer assisted software engineering tools, CASE tools.

Effective tool support for verification and validation is very much dependent on the properties of the chosen conceptual modeling language. This aspect will be discussed in the next chapter.
Chapter 3

A Taxonomy for Model Quality

This chapter proposes a taxonomy for model quality along the same lines as used for software quality. The main purpose is to establish a common terminology with respect to model quality and to identify the factors that provide a given quality.

Section 3.1 reviews previous attempts to define the concept of quality with respect to conceptual models/requirements specifications, indicating several weaknesses and arguing for the need for a better taxonomy. Section 3.2 establishes, based on inspiration from linguistics, the basic terminology of the taxonomy. From the terminology discussion, three types of quality are proposed: syntactic, semantic, and pragmatic quality. The goals and means for each quality type are proposed in Sections 3.3, 3.4, and 3.5, respectively. Finally, Section 3.6 provides some concluding remarks.

3.1 Existing Taxonomies

Only few attempts have been made to identify what quality means in a conceptual model and in a requirement specification. In this section we will discuss the taxonomies of Kung [79], Roman [122], Yeh [160], and Davis [36].

An overview of the taxonomies is shown in Table 3.1 and summarizes the properties for a "good" conceptual model. Based on this overview, the following general observations can be made:

- There is no accepted standard that defines the properties for quality of a conceptual model.
- The properties given in the four taxonomies only amount to a listing of properties that a conceptual model should have. There are no relationships be-
between the various properties, e.g. stating that some properties can be viewed as subproperties of others.

<table>
<thead>
<tr>
<th>Kung</th>
<th>Roman</th>
<th>Yeh</th>
<th>Davis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understandability</td>
<td>Appropriateness</td>
<td>Understandable</td>
<td>Correct</td>
</tr>
<tr>
<td>user-orientation</td>
<td>Conceptual cleanliness</td>
<td>Modifiable</td>
<td>Nonambiguous</td>
</tr>
<tr>
<td>readability</td>
<td>Constructibility</td>
<td>Precise</td>
<td>Complete</td>
</tr>
<tr>
<td>unambiguity</td>
<td>Precision</td>
<td>Unambiguous</td>
<td>Verifiable</td>
</tr>
<tr>
<td>intuitivity</td>
<td>Lack of Ambiguity</td>
<td>Complete</td>
<td>Consistent</td>
</tr>
<tr>
<td>clarity</td>
<td>Completeness</td>
<td>Internally consistent</td>
<td>Understandable</td>
</tr>
<tr>
<td></td>
<td>Consistency</td>
<td>Minimal</td>
<td>Modifiable</td>
</tr>
<tr>
<td>Expressiveness</td>
<td>Analyzability</td>
<td>Abstraction</td>
<td>Traceable</td>
</tr>
<tr>
<td>resolution of detail</td>
<td>Formality</td>
<td>Partition</td>
<td>Annotated</td>
</tr>
<tr>
<td>time perspective</td>
<td>Testability</td>
<td>Projection</td>
<td></td>
</tr>
<tr>
<td>power of expression</td>
<td>Traceability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processing Independence</td>
<td>Executability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>declarative specifications</td>
<td>Incompleteness tolerance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checkability</td>
<td>Modifiable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>concerning validity</td>
<td>Expressive economy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>consistency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changeability</td>
<td>localization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loosely structured</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Four taxonomies for quality of conceptual models.

3.1.1 Kung’s Taxonomy

Kung’s taxonomy is given in the first column of Table 3.1. Although all the properties mentioned are useful, the taxonomy is somewhat confusing.

First, the grouping of the five major model features is not orthogonal. For instance, what is the difference between resolution of detail and unambiguity? If details cannot be resolved, the model must necessarily be ambiguous.

Second, in some cases the grouping itself is questionable. Is unambiguity really a sub-feature of understandability? A model might be ambiguous and still rather easy to understand. Why is validity and consistency considered sub-features of checkability — if anything, it should be the other way around, since checkability is necessary to achieve validity and consistency, which are the features directly requested. Testability, on the other hand, seems to be synonymous to checkability, thus, not adding anything to the taxonomy.

Finally, the taxonomy mixes model features and language features. Expressiveness is something we want from the modeling language in general, rather than from the
3.1. Existing Taxonomies

particular model. Some of the sub-features of understandability could also be considered more associated with the language than with particular models.

3.1.2 Roman’s Taxonomy

Roman’s taxonomy is given in the second column of Table 3.1. Again, this is a rather unsystematic listing of useful features, with some of the same weaknesses as found in Kung’s taxonomy.

First, there is a lack of orthogonality. For instance, precision, formality, testability, and executability are highly interdependent.

Second, there is a mixture of model features and language features. For instance, appropriateness (how well the conceptual constructs of the model fit the nature of the problem) and incompleteness tolerance are clearly properties of the language, not the particular model.

Finally, Roman even mixes model features with method features which is shown by the way he has defined constructibility (the existence of a systematic approach to formalizing the requirements) and testability (the ability of cost-effective procedures for testing if the realization of a component satisfies the requirements). Such properties say more about the development method used than about the model as such.

3.1.3 Yeh’s and Davis’ Taxonomies

Yeh’s and Davis’ taxonomies are given in columns three and four in Table 3.1, respectively.

Yeh and Davis do not confuse model features with language and method features to the extent done by Kung and Roman. Still, they merely provide lists of features, without really trying to systematize the various criteria. Some features of the model, such as correctness, preciseness, completeness, and consistency address the extent to which the model give a correct picture of the requirements. Other features, such as verifiability and modifiability are on a different level — relating to the work you want to do with the model itself, rather than the relationship between the model and the requirements. Yeh comes close to such a distinction by distinguishing between goals and means to achieve these goals. He, suggest abstraction (to suppress details and to concentrate on essential properties), partition (to make the model modular), and projection (to understand a model from different viewpoints) as different means for achieving the goal. However, this distinction could have been analyzed in more detail and the number of means seems somewhat limited.
3.1.4 The Need for an Improved Taxonomy

Summing up the existing contributions to a taxonomy for the quality assurance of conceptual model, it can be seen that lots of valuable model features have been identified and discussed. However, the existing taxonomies are not thoroughly motivated. Typically, lack of a deep analysis of the features involved leads to a lack of orthogonality, a mixture of model features with language and method features, and a mixture of quality goals for the model with the means provided to reach these goals.

By confusing language, model, and even method/tool features, it is difficult to have an idea of what to do next if the quality of a model is to be improved. For quality assurance it is essential not only to know whether the quality is good or bad, but also to identify the reason, so that the modeling languages and development process can be improved to yield better quality in the next project.

In the following we aim at proposing a more systematic taxonomy for the quality of conceptual models. We will emphasize the importance of identifying major quality goals and means/factors for achieving them. Since the modeling language is central in the taxonomy, it is inspired from the relevant discipline of linguistics.

3.2 Basic Terminology of the Taxonomy

Constructing a conceptual model can be viewed as making statements in a modeling language. According to Morris there are three aspects to take into consideration for the study of a language [103]:

- **syntax**: the relations between the signs of the language, without taking their meaning into consideration,

- **semantics**: encompassing the syntax, but also considering the relations between the signs of the language and their meaning (in the world which the statements try to describe), and

- **pragmatics**: encompassing semantics, but also considering the relation between the statements and the user of the language.

These three aspects form the basic relationship in our taxonomy. As indicated in Figures 3.1, syntax relates the model to the language, semantics relates the model to the domain, and pragmatics relates a model to the user. Below, the basic characteristics of language, domain, model, and user will be discussed in more detail – though emphasizing the principles of each term.
3.2. Basic Terminology of the Taxonomy

Figure 3.1: Basic terminology in our taxonomy.

The language

The modeling language — denoted $\mathcal{L}$ — is composed by an alphabet and a grammar. The alphabet contains a set of modeling constructs that are the "building blocks" for the conceptual model. Each construct should have a unique notation. The grammar defines how the modeling constructs are legally combined and consists of a set of grammar rules. The alphabet and grammar are the syntax of the language.

$\mathcal{L}$ can be regarded as a set taken to contain all the statements which are possible to make according to the syntax of the language. For most languages, this set will be infinite. Most of the statements will not make sense with respect to any domain, though.

The domain

The domain — denoted $\mathcal{D}$ — consists of all knowledge that is correct and relevant for developing the information system. The total amount of knowledge can be viewed as an infinite set of correct and relevant statements which could be made about the problem domain at hand. $\mathcal{D}$ reflects the state-of-affairs and will usually not be constant but continuously evolving.

It should be noticed that $\mathcal{D}$ denotes the ideal knowledge about the domain for solving the problem at hand. In some cases, parts of this knowledge is found in organizational documents or kept by people related to the domain. Still $\mathcal{D}$, since it is infinite, is impossible to capture completely.

The model

The model — denoted $\mathcal{M}$ — represents parts of $\mathcal{D}$ by applying $\mathcal{L}$. Also $\mathcal{M}$ can be regarded as a set and can be divided into:
• the explicit model, $M_E$, which is the set of statements explicitly made in the model, and

• the implicit model, $M_I$, which is the set of all statements which only follow implicitly from those of $M_E$.

For instance, assuming that our modeling language is propositional logic and that $M_E$ contains the statement $A \land B$, $M_I$ will contain the statements $A$ and $B$.

Whereas $M_E$ must necessarily contain only a finite number of statements, $M_I$, and thus $M$ itself, will usually be infinite, just like $L$ and $D$.

The user

There are many possible users for a conceptual model: domain experts, buyers/end-users of the information system to be developed, analysts, designers, or even computers (for models which lend themselves to some automatic manipulation). This yields a wide variety of pragmatic concerns. We will restrict the word user to human users of the model, and rather discuss specifically properties which are relevant for automatic processing of the model, since these are often different from what is requested by the user.

Three Types of Quality

Language, domain, model, and user are the basic “actors” in the taxonomy. Now, it is useful to apply these actors to the three language aspects syntax, semantics, and pragmatics. Thus, we obtain three major types of conceptual model quality:

• syntactic quality, i.e. obtaining a model which is correct with respect to the chosen modeling language,

• semantic quality, i.e. obtaining a model which is a correct description of the relevant problem domain, also with respect to syntax (which is encompassed by semantics), and

• pragmatic quality, i.e. obtaining a model which is good for the users of the model.

As will be discussed throughout this chapter, pragmatic is taken to mean only those features which are exclusively pragmatic, i.e. not including semantics or syntax.
3.3. Syntactic Quality

It important to be aware of the difference between semantic quality and pragmatic quality. With respect to the distinction between the full model $\mathcal{M}$ and the explicit model $\mathcal{M}_E$, we can elaborate further on the difference. As indicated in Figure 3.2 semantic quality affects choices between alternative models $\mathcal{M}, \mathcal{M}'$ — which say different things about a domain. Pragmatic quality, on the other hand, only affects choices between alternative explicit models, i.e. between many different ways of saying the same thing. Thus, the model $\mathcal{M}$ is already given, and pragmatics only address the choice between various possible $\mathcal{M}_E$.

We will in the following discuss the three quality types in more detail. We aim at grouping the different model errors and establishing the quality goals for each type. Moreover, we will identify the various means for obtaining the different quality goals.

3.3 Syntactic Quality

Syntactic quality means the absence of errors in the model with respect to the language.

Syntactic Goals

There is only one syntactic goal, syntactical correctness, meaning that all statements in the model are according to the syntax of the language, i.e.
\[ \mathcal{M} \setminus \mathcal{L} = \emptyset \] (3.1)

The corresponding error, \textit{syntax error}, occurs when the above equation does not hold. Syntax errors are caused by either:

- \textit{morphological errors}: i.e. the model expresses statements with symbols that are not defined in the alphabet of the language.

- \textit{syntactical incompleteness}: i.e. the model lacks constructs or information to obey the grammar of the language.

\section*{Syntactic Means}

To check syntactical correctness, the modeling language must of course have a \textit{precisely defined syntax}. To include tool support for syntax checking, the modeling language must have a \textit{formal syntax} that can be manipulated by a computer.

\section*{3.4 Semantic Quality}

Semantic quality means the absence of errors in the model with respect to the domain.

\section*{Semantic Goals}

Our taxonomy contains two semantic goals; \textit{validity} and \textit{completeness}.

\textbf{Validity} means that all statements made by the model are correct and relevant with respect to the problem domain, i.e.

\[ \mathcal{M} \setminus \mathcal{D} = \emptyset \] (3.2)

The corresponding error class is \textit{invalidity}, meaning that the above equation does not hold.

\textbf{Completeness} means that the model contains all the statements which would be correct and relevant about the problem domain, i.e.
3.4. Semantic Quality

\[ \mathcal{D} \setminus \mathcal{M} = \emptyset \] (3.3)

The corresponding error class is *incompleteness*, meaning that the above equation does not hold.

Remember that \( \mathcal{M} \) does not only contain the explicit statements of the model, but also the implicit ones, i.e. we do not have to state explicitly every correct statement of \( \mathcal{D} \).

So, in our taxonomy only two basic semantic quality goals are defined. This is a rather dramatic reduction of semantic-related properties stated in the taxonomies in Table 3.1. Some of those are subsumed by validity and completeness:

- **Minimality (Yeh)** or the similar processing independence (Kung) are directly subsumed by validity. If \( \mathcal{M} \) contains a statement which "overconstrains the design of the system", this statement is simply invalid, i.e. not part of the ideal domain knowledge \( \mathcal{D} \).

- **Annotation (Davis)** means to associate priorities to various requirements. Such priorities are clearly domain knowledge, and thus, lack of annotation is simply a case of incompleteness — if priorities of requirements are relevant to the modeling task at hand.

- **Traceability (Roman, Davis)** means that there should be couplings e.g. between a requirements specification and a design, illustrating which requirements are satisfied by what parts of the design. Since the model itself becomes a part of \( \mathcal{D} \) as soon as the modeling activity starts, this knowledge can also be considered relevant domain knowledge just like anything else, and thus, lack of traceability is a kind of incompleteness (for problems where traceability is relevant).

- **Consistency (Kung, Roman, Yeh, Davis)** is a very frequently stated semantic goal. However, we will here show that it is actually subsumed by validity and completeness. The error class *inconsistency* means that the model contains contradictory statements, i.e.

\[ s, \neg s \in \mathcal{M} \] (3.4)

In the strict sense of propositional logic, an inconsistency will imply that every possible statement can be deduced from a pair of contradictory statements or \( \mathcal{M} = \mathcal{L} \). This is clearly a special case of invalidity, since \( \mathcal{D} \) is not taken into consideration. In the real world, however, it does not seem reasonable to allow the deduction of everything once we have a pair of contradictory statements — after all the real world is not always logical — for instance, many organizations do have contradictory business rules. In this case, there must be some (explicit or implicit) *meta* statement in the organization to resolve the contradiction. Thus, it is possible to imagine a situation
where we can have two contradictory statements which are both valid, a
*normal* rule and an *exception* rule. In fact, this is a case of incompleteness —
because the meta statement would have allowed the two statements to exist
side by side has not been included in the model.

So, in the case of inconsistency, either

- there is invalidity, and by removing the invalid statements from $\mathcal{M}$,
  inconsistency would also disappear, or
- there is incompleteness, and by adding some missing statements to $\mathcal{M}$
  the model will again be consistent.

From the above argumentation, inconsistency is subsumed by invalidity and
incompleteness and we do not consider it an error class in its own respect.
However, consistency is much easier to check than validity and complete-
ness. Often it is cost-effective to obtain full consistency. Thus, from the view-
point of tool support, it will often be convenient to consider inconsistency
as a special error class.

- Lack of ambiguity (Kung, Roman, Yeh, Davis) is another goal which is often
  listed. The corresponding error class *ambiguity* means that the model can be
  interpreted in two (or more) alternative ways. Our set $\mathcal{M}$, as defined above,
  will be the union of these two interpretations, i.e. $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2$. Either
  these are contradictory, leading to the same argument as for inconsistency
  above, or they are not. If the whole of $\mathcal{M}$ is consistent and valid, there does
  not need to be anything wrong with the ambiguity at all — except maybe
  that it should have been stated explicitly that both interpretations $\mathcal{M}_1$ and
  $\mathcal{M}_2$ were intended — which is again a case of incompleteness. Thus, any
  case of ambiguity must also necessarily be subsumed either by invalidity or
  incompleteness, and we will not consider it a separate error class.

**Semantic Means**

Generally, validity and completeness cannot be checked without involvement from
humans — since these goals refer to the relationship between the model and the
real world (which is beyond computer processing). However, the more limited,
subsumed goal of consistency can sometimes be checked automatically. This re-
quires that the language has formal semantics, and that there is tool support for
deduction.

To achieve completeness, it is also important that the modeling language is suffi-
ciently expressive. *Expressiveness* (Kung) or the similar *appropriateness* (Roman) is
sometimes presented as a model issue, but is actually a language issue: if

$$\mathcal{D} \setminus \mathcal{L} \neq \emptyset$$

then the language is not sufficiently expressive for the problem at hand. From the
model's point of view, however, this will simply lead to incompleteness.
Expressiveness could be seen to include both formality (Yeh, Davis), i.e. the ability to express formalized knowledge, as well as informal or fuzzy knowledge. Thus, lacking support for formality or informality is a language issue — again, from the model's point of view, the observed result would be incompleteness.

An important aspects of expressiveness is executability which in particular addresses the dynamic properties of a domain. Executability presupposes formality and can be of great help for ensuring validity and completeness, by allowing prototyping. This will be thoroughly discussed throughout the thesis.

3.5 Pragmatic Quality

Pragmatic quality addresses the relationship between the model and its users. Whereas the criteria for semantic quality prescribe the choice of one particular model $M$ which is the “right” one, pragmatic quality criteria take this particular “right” model for granted, thus only prescribing the choice of one particular explicit representation $M_E$ of $M$, instead of any other explicit representation which is somehow not as good. However, following from the discussion of “wicked problems” in Section 2.2, it is impossible to operate in practice with absolute “rights”. Thus, the distinction between pragmatic goals and pragmatic means becomes rather important: pragmatic goals may affect the choice of $M$ itself, whereas pragmatic means deal with language, method or tool issues, or with the choice of alternative explicit models.

Pragmatic Goals

The only pragmatic goal in our taxonomy is feasibility that is not discussed by any of the taxonomies presented in Section 3.1. Since total validity and completeness are impossible to achieve, too hard attempts to satisfy these criteria will make the modeling activity prohibitively expensive or time-consuming. So, since feasibility modifies the choice of $M$ with respect to the idealized and impossible requirements of validity and completeness, it can be divided into:

- **feasible validity**: $M \setminus D = A$ but there is no statement $s$ in $A$ such that the value added to the conceptual model by removing $s$ from $A$ exceeds the cost of the work necessary to find out that $s$ is invalid — or alternatively: the customer is not willing to pay for efforts to further increase the validity of the model.

- **feasible completeness**: $D \setminus M = B$ but there is no statement $s$ in $B$ such that the value added to the conceptual model by including $s$ in $B$ exceeds the cost of capturing $s$ — or alternatively: the customer is not willing to pay for efforts to make the model more complete.
Feasibility relax the more idealistic goals of semantic quality and it has to balance a trade-off between the costs and values for achieving a given model quality.

Both feasible validity and feasible completeness are goals that are defined by factors of individual development project. Clearly, the competence of the model builders [1, 46, 148] (i.e. analysts, end-users, domain experts) is a major factor to achieve highest degree of validity and completeness within the allocated costs. Here, we focus on pragmatic means that are determined by the user’s support environment.

Pragmatic Means

Although feasible validity and feasible completeness relax the more idealistic semantic goals, the semantic means are clearly important for establishing the pragmatic goal as well. Since language expressiveness encompasses properties like formality and executability that are directed towards the computer, we will not introduce formality and executability as pragmatic means in their own right. We will direct the pragmatic means towards the user of the model and concentrate on the following factors: comprehensibility, modifiability, and modeling freedom.

Comprehensibility is a problematic issue because:

- There are a wide variety of users of the model (end-users, domain experts, developers, ...) and these have different insights in the problem domain, as well as varying degrees of experience with conceptual modeling — creating a wide array of partly contradictory criteria for comprehensibility.

- With a big model, no single user is expected to comprehend everything — each one will be mainly occupied with some restricted part of the model. This division of interest between various users may be both vertical (different levels of abstraction — one only interested in a rough overview, another one interested in details) and horizontal (different users interested in different parts of the domain).

Recognizing these problems, comprehensibility should not merely consider the model “as it has been recorded”. As will be further discussed in Chapter 5, there exists different approaches for improving model comprehensibility. All approaches are concentrating on generating a wide variety of alternative explicit models and projections, in order to satisfy various criteria for comprehensibility.

A direct language criteria to provide comprehensibility often referred to in the literature is visualizability making use of graphical notation to reveal complex structures of the domain. Different mechanisms like hierarchies, encapsulation, connectness, and adjacency have been found useful [64]. Other
features, such as size, shape, and color can also be exploited to improve perceptibility. Visualization is promoted by Harel [65] as one of the major means to improve the quality of information systems. In Section 4.1 a brief survey of visual modeling languages will be given.

Modeling freedom has been discussed by Feather [44]. Although error-free models are the major goal of model quality assurance, some errors can be advantageous to have on a temporal basis. Too much focus on model quality might hamper the creativity of the modeling process. Of the four taxonomies discussed in Section 3.1, only Roman addresses it (incompleteness tolerance). Validity and completeness represent ideal requirements for a “finished” model. However, in the initial phases of modeling, it is useful or even necessary for the analyst to be able to produce temporary models which are highly invalid or even syntactically wrong.

For instance, given a graphical language consisting of “boxes”, “lines” and a syntactical requirement stating “all boxes should be linked to a line”. It may be difficult to draw a box and a line at the same time — thus, some temporary incorrectness has to be accepted. Generally, some degree of freedom is necessary for the modeler to be able to work effectively. Of course, modeling freedom is not a feature of the model, but mainly a requirement to the tool support.

Modifiability is necessary with respect to maintaining validity and completeness of a model over time. The importance of modifiability will thus depend on the purpose of the model — and of the properties of the domain. If the domain is rapidly evolving and the model is supposed to be in use for a long time, it is very important. On the other hand, if the domain is only evolving very slowly or the model is not supposed to be in use for long, it is not very important.

Modifiability is an important factor in the change processes of conceptual modeling and model quality assurance. As already stated though, management of the change process is not addressed in this thesis.

3.6 Using the Taxonomy

The proposed taxonomy for the quality of conceptual models is summarized in Figure 3.3. We have shown the basic quality goals, together with means which can be used to achieve these goals. The taxonomy is not meant to be completely comprehensive but rather to establish qualitative relationships between goals and means. As will be shown throughout the thesis, the taxonomy can be further expanded with new means/criteria particularly for achieving pragmatic quality.

Verification and validation can be related to the taxonomy as follows:

- The syntactic quality of the model is addressed by verification.
Figure 3.3: Summarizing the taxonomy.

- The semantic quality of the model, is mainly addressed by validation. The simplest problem of semantic quality assurance, consistency checking, can be addressed by verification. In some cases, user involvement may be required to resolve inconsistencies (e.g., inconsistent business rules in the organization); in other cases, developers can fix the error without asking users (e.g., checking the consistency between a design specification and the corresponding requirements specification, in which case an error would usually mean that the design is wrong).

- The pragmatic quality of the model is addressed by validation (and cost-benefit analysis).

The framework gives no concrete guidelines on how to do verification and validation. Different approaches for verifying and in particular for validating a model will be discussed throughout the thesis relying on the properties established in the taxonomy.

It should be noted that the taxonomy aims at ordering the goals and means in a systematic way. A discussion of the optimal set of means is outside the scope of this thesis. However, it should be noted that some means are contradictory and that trade-offs have to be made. A major trade-off — feasibility considerations — is already inherent in the framework. Moreover, expressiveness/formality is a semantic mean to obtain validity and completeness. However, formal and complex model may be incomprehensible which will hamper an easy validation of the model. Approaches for improving model comprehensibility are discussed in Chapter 5.

Although our taxonomy does not prescribe a way of working, a modeling language must be chosen before we can be able to write anything at all. However,
as stated earlier, the modeling language must be appropriate for the problem at hand. To make such a choice, we have to have some knowledge about the domain to be modeled before modeling starts. In the next chapter we will give a brief survey of modeling languages emphasizing the properties of expressiveness and visualization.
Chapter 4

Conceptual Modeling Languages and CASE tools

In the previous chapter the proposed taxonomy is established on a general level. No particular language or tool was presented or discussed. This chapter will therefore give a brief flavor of the versatility of modeling languages and CASE tools. Since the number of modeling languages and tools is large, probably several hundred, a comprehensive survey is impossible within the scope of this text. Besides, several language surveys already exists e.g. in IFIP WG 8.1 CRIS Series [108, 106, 107] and in [36, 132]. A thorough overview and evaluation of commercial CASE tools are found in [40, 119].

We will therefore concentrate on development trends of languages and tools, supplemented with some examples. The languages and tools are presented in Section 4.1 and Section 4.2, respectively. Section 4.3 discusses the development trends with respect to model quality assurance.

4.1 Conceptual Modeling Languages

In the following presentation, we restrict ourselves to visual languages mainly because:

- visualization is a central pragmatic mean to achieve comprehensible models.
- most CASE tools employ visual languages, and
- they form the basis for the languages in the PPP environment.
In the sequel we will order the languages in the traditional classification schema [70, 109]: data-oriented, process-oriented, and behavior-oriented. The schema divides the languages according to their expressiveness, or rather to what perspective of the domain they are meant to model. Moreover, we discuss contemporary efforts emphasizing integration of languages in order to further improve their expressiveness.

4.1.1 Data-Oriented Languages

Data-oriented modeling languages are used to describe the static aspect of a problem domain. The first data-oriented languages were inspired from the database realm. The Entity Relationship (ER) language ([28]) is a well established static language. It views the problem domain as a "snapshot" of classes of entities and their relationships. In Figure 4.1a the statement "a customer has an account" is modeled using the original ER notation. Because of its intuitive use and its straightforward mapping to the relational data model [145], the ER language has gained considerable popularity, both in industry and research communities.

![ER Diagram](image)

**Figure 4.1:** Four static modeling languages with different expressiveness.

However, the expressiveness of the original ER language is limited and in the Extended ER (EER) language ([139]), attributes are introduced to describe the properties of the entities and relationships, whereas cardinality specifications restrict the scope of the relations. In Figure 4.1b these aspects are shown. A customer has the attributes cust_no, name, and address. Moreover, the cardinality between
customer and account expresses the statement "a customer can have many accounts".

Still, the expressiveness of the above languages only results in a “flat” view of the domain. Structure-oriented languages provide modeling constructs for representing the hierarchical structures of a problem domain [114]: classification, generalization, aggregation, and association. SHM+ ([21]) provides all these structures and generalization in this language is shown in Figure 4.1c. An account may be a generalization of salary account, savings account, and loan account. Moreover, aggregation in TEMPORA’s ERT language ([94]) is shown in Figure 4.1d. Here, the complex object models the complex attribute address which is composed of street, city, and country.

A thorough evaluation of data and structure-oriented modeling languages is found in [114].

Figure 4.2: A DFD model of parts of a banking domain.
4.1.2 Process-Oriented Languages

Process-oriented modeling languages are used to model those parts of a domain that concern business activities/functions and their exchange of data/information. Most languages are rooted in functional decomposition, where complex activities are divided into subactivities. The decomposition continues to a point where the activities are intellectually manageable. One of the most popular process-oriented modeling languages is the Data Flow Diagram (DFD) language ([38, 49]). In Figure 4.2 parts of a banking domain is modeled using Gane and Sarson's notation [49].

Although the DFD language has gained high popularity, it suffers from several shortcomings:

- It has no constructs for stating when a process is triggered or terminated.
- The content of a flow is limited to data only and is not explicitly modeled. Control flows and material flows are as likely as data flows in interprocess communication.
- The limited expressiveness leads to several valid interpretations of a DFD model. For instance, the termination of P1.1 in Figure 4.2 can be done in three different ways:
  1. both Withdraw transaction and Deposit transaction are sent,
  2. either Withdraw transaction or Deposit transaction is sent, or
  3. either Withdraw transaction or (exclusively) Deposit transaction is sent.

Some of these shortcomings are eliminated in the Transformation Schema ([151]) that in particular allows for control flows and processes¹. As will be further discussed in Section 8.2, the shortcomings of DFD is a major motivation for the PrM language in the PPP environment.

4.1.3 Behavior-Oriented Languages

Behavior-oriented modeling languages address the dynamics of a domain explicitly. Typically, a domain is regarded as a set of states and transitions between the states. State transitions are triggered by events. These constructs are found in State Transition Diagram (STD) ([72]).

Petri-Net ([115]) is another well-known behavior-oriented modeling language. A model in the original Petri-Net language is shown in Figure 4.3. Here, places indicate a system state space and a combination of tokens included in the places

---

¹Processes are called transformations in the Transformation Schema [151].
determine a specific system state. State transitions are regulated by a firing rule that is described as follows [115]:

A transition is *enabled* if each of its input places contains a *token*. A transition can *fire* sooner or later when it is enabled. After the firing of a transition, a token is removed from each of its input places and a token is placed into each of its output places.

Moreover, the figure shows how dynamic domain properties like *asynchronization* (*precedence*), *concurrency*, *synchronization*, *exclusiveness*, and *iteration* are modeled in Petri-Net. The associated model patterns along with the firing rule above establish the *execution semantics* of a Petri-Net.

Figure 4.3: Different dynamic properties modeled by the original Petri Net language (from [132]).

In recent years several extensions of the original Petri-Net language has been proposed. Generally, the expressiveness is improved and tailored to specific domains. For instance, **Timed Petri Nets** ([100]) provide probability distributions that can be assigned to the time consumption of each transition and is particularly suited for *performance modeling*. **BNM** ([131]) introduces an extended firing rule using *pre-* and *postconditions* to further define the transition criteria. Moreover, the dynamic part is linked to a structural part. As such, the language can be used to model database applications. The structural part describes the database
content, whereas the dynamic part can be used to model database transactions.

The execution semantics of the STD language and the Petri-Net language assumes that transitions takes zero time. Thus, the languages are not suitable for decomposing transitions into subtransitions. **Statecharts** ([63]) is an extension of STD and provides hierarchical modeling constructs that eliminate these shortcomings.

In Figure 4.4 the functions of a digital watch are modeled by Statecharts. The different states of the watch are represented as well as the events that causes the transitions. For instance, given that the watch is showing time (is in state time), the state alarm is entered by the event a (pushing the mode switch button). The hierarchical features of the watch are in Statecharts expressed as AND and XOR decompositions. Since the functions display and light of the watch can be active simultaneously, their associated states are modeled as an AND decomposition. On the other hand, light can either be on or (exclusively) off, and their associated states are modeled as an XOR decomposition.

An evaluation of some behavior-oriented modeling languages is given in [35].

### 4.1.4 Other Language Perspectives and Integrated Languages

Although the expressiveness of the languages presented above are improved in recent years, languages from only one perspective are too restricted to build model of feasible completeness. To further extend the expressiveness, a major trend has been to integrate languages from different perspectives. Thus, the expressiveness of the integrated language enable the developer to model a larger part of the relevant domain knowledge than by using a "stand-alone" language.

Examples of languages that in different ways integrate the above perspectives are
4.1. Conceptual Modeling Languages

TEMPORA ([138]), BNM ([131]) and REMORA ([121]).

As part of striving for increased expressiveness and formality of the language, some conceptual modeling languages combines a visual notation with logics. For instance, ERAE ([61]) combines an extended ER notation with first-order typed temporal logic version. Furthermore, the ERL language in TEMPORA ([94]) is a semi-natural language for explicitly modeling business rules and is combined with PID (a DFD-like language) and ERT (an extended ER language that have constructs for time stamping of “static” aspects of the domain) [138].

Object-orientation is of growing popularity in IS engineering and object-oriented modeling languages have emerged from object-oriented programming languages. Object-oriented modeling languages emphasize the hierarchical structures of object classes in the domain objects. Inheritance and encapsulation of static knowledge are central aspects of the language. Moreover, message passing enables communication between objects and controls the behavior of the relevant objects by activating their methods. Examples of object-oriented modeling languages are OMT ([124]) and OOA ([30]).

AMADEUS ([95]) is a result of a major integration effort. Its modeling constructs are compiled from 10 modeling languages. In Figure 4.5 the meta model of the AMADEUS language is shown. The alphabet of the language and the basic structural grammar are expressed by an ER-like notation. Subsequently, to establish a meta model, a data/structure-oriented language is used to model the modeling constructs of language.

Meta-modeling is a rapidly growing area in IS engineering and serves several interesting purposes [142]. First, a meta model can be used to explicitly assess the expressiveness of a modeling language. Thus, the expressiveness of different lan-
guages can be compared by viewing different meta models. Second, a joint meta model, as the one in Figure 4.5, can be used as a kernel so that translation between models in different languages can be effected [95]. Finally, a meta model forms the basis for a CASE repository.

4.2 CASE tools

During the last decade there has been a dramatic increase in the number of tools supporting different aspects of the IS engineering process. Below, we will give a brief overview of the “CASE tool revolution”. The presentation will be ordered in the first and second generation [50]. The major characteristics of each generation will be outlined. Various tools from each generation will be mentioned and thorough descriptions of them are found in [68, 119, 40].

4.2.1 The First Generation CASE Tools

The first generation CASE tools emerged in the early eighties, and consisted of so called point tools. Thus, the tools only supported a limited number of tasks within a specific development stage. Two main tool groups belong to this generation: Analyst workbenches and code generators.

Analyst Workbenches exploited the emergence of graphical work stations and personal computers. Their main support were concentrating on helping the developer in drawing models by means of a computer instead of "pen and paper". Exce1erator and Software Through Pictures were some of the early workbenches and provided modeling support for ER ([28]) and DFD ([38, 49]). As part of the modeling support restricted syntax check were provided as well.

Code generators were concentrated on generating application code. Generally, the generators provided the necessary administration statements which were necessary to execute large applications. Database definitions were generated from ER-like models, whereas procedure templates were generated from dynamic models and module hierarchies. The remaining process logic was often left for programmers to fill in. Install/1 and Spectrum are tools that belongs to this category.

4.2.2 The Second Generation CASE Tools

As different point tools were developed to support single, different tasks, a next step was to make them cooperate. The second generation CASE tool, which emer-
4.2. CASE tools

ged in the late eighties, emphasizing integration of tools. The CASE repository plays a major role in this respect and allows for storing specifications from different CASE components. As such, a repository permits the sharing of specifications among the integrated CASE components.

Three main integration architectures have been proposed [119]: CASE framework, Integrated CASE tools, and Integrated Project Support Environment. Furthermore, a fourth CASE architecture, CASE shells, emerged during the second generation. In the following, the characteristics of the four architectures are explained.

**CASE framework** is an open architecture which allows for integrating different point tools to end up with a tool set covering the whole development life-cycle. A characteristic property of this approach is to obtain a standard tool interface so that various tools from different vendors can be integrated. Although no common standard has been established, proposals like ANSI/IRDS, ISO/IRDS, and AD/cycle have been suggested [119].

**Integrated CASE (ICASE) tools** aims at supporting one well-defined approach throughout the whole development process. As such, ICASE tools emphasize phase integration be letting the output of one phase being direct input to the next. The integration is enabled by a joint meta model for the supported approach. Some central commercial ICASE tools are IEF, IEW, Foundation, and Oracle/CASE. The number of language that are supported in these tools are high (10 to 20). The verification of consistency between the associated models is restricted to using cross-referencing matrices [68].

**Integrated Project Support Environment (IPSE)** is a framework which shares several characteristics with the CASE framework and ICASE, such as 1) integrating point tools with a uniform user interface, 2) supporting the whole life-cycle, and 3) providing flexibility and method independence. Additionally, IPSEs includes administrative support functions like configuration management, multi-user support, and access control. Typical environments having these features are Maestro, ISEE, and ADE.

**CASE shells** recognize that effective implementation of CASE tools in organization is very much dependent on customization of the tools. By providing a meta model and even a meta method, the CASE shell architecture aims at tailoring a CASE tool and its functionality to a user defined approach. Although some commercial CASE shells with restricted functionality exists in the market, CASE shells are still mainly a research topic accompanying the area of meta-modeling [142].
4.3 Towards Model Quality Assurance in CASE Tools

As stated in the introduction to this chapter, a comprehensive survey of languages and tools is outside the scope of this text. Emphasizing the development trends of modeling languages and CASE tools, the following observations can be made:

- Modeling languages within contemporary CASE tools are visual. However, the number of languages employed in CASE environments is high and the languages are usually loosely coupled. Moreover, typical CASE languages like DFD, ER, etc. have a limited expressiveness.

- Modeling languages with higher expressiveness have only to a limited degree been adopted in commercial CASE tools. One of few exceptions is STATEMATE ((66)) which supports Statecharts ((63)) in order to develop real-time systems.

From the argumentation in Chapter 3, expressiveness is a central language property for model quality assurance. Consequently, the limited support for model quality assurance in most contemporary CASE tools stems from an insufficient language basis. Also, when the number of languages are high, there tends to be an information gap between the various parts of the models making the formalization of the integration difficult.

So, model quality assurance in future CASE tools should be built around expressive modeling languages.

Although increased expressiveness is a major prerequisite for advanced verification and validation, a result of using expressive languages can be incomprehensible models. This will hamper the validation process. In the following chapter, three approaches for improving model comprehensibility are described.
Chapter 5

Three Validation Approaches

In Section 3.5 we claimed that comprehensibility is of major importance for validating a conceptual model. Despite the user-oriented nature of many conceptual models, they become increasingly difficult to read and fully understand when their size and complexity grow. Also, as the modeling languages have become more expressive and formal, models are difficult to judge only by inspecting the model.

This chapter outlines three approaches for making a conceptual model more comprehensible. The Complexity Reduction Approach and the Presentation Approach are presented in Sections 5.1 and 5.2, respectively. A combination of the approaches is briefly discussed in Section 5.3. The approach which is particularly addressed in this thesis is the Prototyping Approach. Section 5.4 presents the baseline for this approach.

5.1 The Complexity Reduction Approach

When a conceptual model is large and complex, the users may get confused by all of the details. By reducing the complexity of the model, it is envisaged that the comprehensibility of the model will improve and make the validation process more feasible.

The approach deals with this problem either by grouping detailed information, focusing on relevant views of the model, or improving the layout of the model. Consequently, we denote the techniques for reducing model complexity, structuring techniques, abstraction techniques and layout techniques. We will discuss the techniques in the context of visual modeling language. The approach has its textual counterpart in hypermedia and document layout. To illustrate the characteristics of each technique, the example model in Figure 5.1a will be used as basis.
A further description of this approach is given in [126].

**Structuring Techniques** aim at grouping together patterns that conceptually belong together. Typically the techniques exploit features like hierarchy, encapsulation, connectedness, adjacency of the modeling language [63]. These features are combined with the style used to structure the model. The tree-style and the onion-style are the most common ones [129]. Assuming that arrows in our example model mean “superset of”, B and its subsets are structured by a tree-style in Figure 5.1b and by an onion-style in Figure 5.1c.

Of the languages discussed in Section 4.1, the tree-style is often exploited in newer semantic data languages like SHM+ ([21]). The onion-style, on the other hand is used in DFD ([38, 49]) and Statecharts ([63]). Also, the complex objects of ERT ([94]) aggregates static properties in the onion style.

Although the structuring techniques often are inherent in the language, they can be regarded as a useful add-on to an initial flat modeling languages like Petri-Net ([115]) and ER ([28]). This is further elaborated in [126].

**Abstraction techniques** A model may be difficult to read because of irrelevant details. Abstraction techniques address the problem by focusing on the relevant
5.2. The Presentation Approach

part of the model (a view), whereas the irrelevant information is removed (ab-stracted away).

The model in Figure 5.1b is a view of the original model emphasizing B and its children. A and G are abstracted away.

Whereas structuring techniques addresses the "conceptual complexity" of a model by providing structural language features, abstraction techniques "hide" specificational details. Abstraction techniques is therefore more an additional feature — which can be supported by tools — than a basic language feature.

Layout Techniques  A conceptual model is a result of a development process, and has been analyzed and modified iteratively during the model's construction. Thus, the model may often become more complicated than it ought to be. Layout techniques aims at simplification by changing the layout of the model, by redrawing or rewriting the elements of the model in a more perceptible way.

In order to automate the layout techniques, graph layout algorithms [39, 134, 136] to avoid/minimize crossing lines, minimize the drawing area of the model, etc. maybe exploited. In Figure 5.1d the layout of our example model is improved so that crossing lines are avoided. The layout techniques can of course be used in combination with other complexity reduction techniques. This is further elaborated in [126].

5.2  The Presentation Approach

Many conceptual models are difficult to comprehend because of the user's unfamiliarity with the modeling language. The syntax and the semantics of formal modeling languages may easily confuse unskilled users, in particular if the concepts of the languages differ from what the user is acquainted with. Presentation techniques try to alleviate this by presenting the model in some other language with which the user is more familiar. Two major presentation techniques are paraphrasing and model translation.

Paraphrasing  is a well established technique that, as indicated in Figure 5.2a, generates natural language expressions from (parts of) the conceptual model. There is a one-to-one correspondence between model and sentence, and a sentence will usually not depend on other sentences in the paraphrase.

Systems that exploit the paraphrasing approach have been reported in [61, 73, 105, 120, 135]).
Figure 5.2: Two presentation techniques.

Model translations Although paraphrasing translates the model content into natural language sentence, it sometimes feasible to preserve the model formality, when presenting the content in a syntax the user is more familiar with. So, the translation algorithm is assumed to preserve the semantic content of the model, but to present it in a more user-friendly language. The target modeling language should be a language that is easier to read than the original language. In Figure 5.2b, we have translated a DFD model in DeMarco/Yordon’s notation [38] to a DFD model in Gane/Sarson’s notation [49].

Systems that provide translations between formal languages have been reported in [37, 47, 104, 105]. In ARIES [73], a powerful internal knowledge representation language is used, and from this representation one may define translations to both natural languages and other formal languages.

5.3 Towards Combined Approaches

The two approaches above are motivated from different viewpoints. The first addresses the model complexity, whereas the second addresses the model familiarity. Although the techniques within each approach can be exploited separately,
techniques within and across the approaches can be *combined* to improve the model comprehensibility.

Typically, the three complexity reduction techniques can be combined in many different ways. This is further discussed in [126].

*Explanation generation* is a combined technique that generates natural language text from conceptual models. In contrast to paraphrasing, explanation generation allows the user to ask questions about the model. When generating an appropriate answer, the generator may take into consideration the characteristics of the user as well as the context in general. The quality of the text may be improved by including instances from the model, and traces from its execution, in the explanation generation process. Even explicit knowledge about the concepts of the language may be put into the generation process to make the explanations more understandable and structured. Explanation generation is further discussed in [57].

The number of combinations may be extended by introducing the prototyping approach. This is further elaborated in [155]. In the remaining parts of this thesis we will concentrate on prototyping as a stand-alone approach.

### 5.4 The Prototyping Approach — Baseline

Since conceptual models are constructed early in the development process, it is difficult to judge the consequences of the model before it is realized in the final information system. The prototyping approach exploits the *executability* of conceptual modeling languages and translate the associated models to an executable prototype. A prototype is a tangible software product which can be used to validate the dynamic properties of a model.

Before we establish the prototyping approach in a CASE tool context, we will in the remaining of this chapter provide the baseline for the approach concentrating on *prototyping* and *translations*.

### 5.4.1 Prototyping

Prototyping emerged in the early eighties as an alternative/supplement to the rigorous phase-oriented way of developing information systems. It is a basic component in the *spiral model* [19] 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 [3].
In the literature, two major directions are frequently referred: throwaway prototyping [54, 53] and evolutionary prototyping [19, 32, 101, 149]. 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 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 reflect on the external appearance of the system (screen, reports, dialogues) with limited or no functionality [149]. 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 [32]. However, screen pictures may be expanded with some functionality so that the user interface in operation can be validated. This effect is exploited in Wasser- man’s USE methodology [152].

**Experimental prototyping**  is introduced by [48] 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 of appropriate hardware for the system (hardware prototyping).

5.4.2 Prototyping — A Taxonomy

These prototyping approaches mainly address how to rapidly create a prototype, 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 domain knowledge/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 [123] and relates the prototype to the model, whereas the last two aspects are easily deduced from literature, and relates the prototype to the time schedule of software 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 [123]. 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 [123].

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 can live during the development process before it is discarded. Throwaway and evolutionary prototyping have short and long durability, respectively.

Some of these aspects will be further addressed in connection with the requirements to the prototyping environment in Section 6.4. A further discussion of this prototyping taxonomy is found in [87].
5.4.3 Towards Translational Prototyping

In the literature the term rapid prototyping is often linked to programming in very high level languages (VHLL). These are powerful languages intended to provide high programming productivity. In industrial practice VHLL is known as 4th generation language that is particularly tailored to prototype database applications. In spite of high programming productivity, they suffer from shortcomings like:

- *Their scope of applications is limited.* The language expressiveness is insufficient if problems are outside their scope. 4GL’s only support a data-oriented view of applications.

- *There is a lack of development methods.* This leads to problems in coping with the complexity of large domain.

In our approach, we have stressed the value of having a visual expressive conceptual modeling language available in order to structure and reveal the domain knowledge. The challenge is to see whether the knowledge is reflected in an operational system.

![Diagram](image)

Figure 5.3: “Manual” Prototyping vs. Translational Prototyping.

Given an available conceptual model, two main alternatives for prototyping exist: “manual” prototyping and translational prototyping. These alternatives are indicated in Figures 5.3a and 5.3b, respectively. The first alternative emphasize building a VHLL prototype by developing a program based on the content of the model. The second alternative aims at automatically translating the model into a prototype in some programming language (VHLL or 3GL).

From the above shortcomings, we reject the “manual” prototyping approach. Furthermore, since the prototyping approach is meant to improve the comprehensibility a “manual” programming effort may be hampered due to model incomprehensibility. Also, if the model contains sufficient specification details to be executable it is unnecessary to make a new program, as the model already has the functionality of a computer program.

We will therefore explore the translational prototyping approach in more detail.
5.4.4 Translations

This section establishes the baseline of model translations. To discuss the details of the translation process, we use a traditional compiler architecture, which will form the basis for building translation assistants for CASE tools.

Translation vs. Transformation

We shall clarify a couple of terms. So far, we have consequently used translation to denote a mapping between different type of representations. In the literature, however, transformation is most commonly used [117]. To avoid possible misunderstandings, our interpretation of these terms is informally explained as follows:

- *Translation* means that an input (source) representation in language L is converted to an output (target) representation in another language L'.

- *Transformation* means that an output representation in language L is converted to an output representation in the same language L.

The key difference between translation and transformation is whether the output representation is in the same language as the input representation or not. As shown in Figure 5.3b, the mappings in the prototyping approach is of translation type. Furthermore, recalling from Sections 5.1 and 5.2, transformations are mainly used in the complexity reduction approach, whereas translations are exploited in the presentation approach.

Automatic Programming

The translation activity in the prototyping approach is inspired from automatic programming [9, 10] and addresses the labor intensiveness of software development. The ultimate goal of automatic programming is (quoted from [2]):

"...to simply state the requirements to the future systems and automatically translate these statements into a complete and runnable program."

The underlying idea of automatic programming is to replace some time consuming and error sensitive development task with a translation assistant. In this way, the developer can concentrate on more creative tasks. To build such an assistant, it is required that the task to be automated is recorded and formalized [10]. Since the major role of the assistant is to translate one representation to another one, well
defined rules of the translation process must be established as well as the format of the input and output representations. When an assistant has been developed, it can be given the responsibility of performing the dedicated tasks.

Design and Domain Knowledge

From a knowledge engineering point of view, a translation assistant has to consider two types of knowledge [117]: Domain knowledge and design knowledge. Both types can be formalized and incorporated in the assistant. Design knowledge comprises various design tasks, as well as general development principles. Also, it covers experiences and heuristics of individual expert developers stating advantageous ways of performing specific tasks.

The following statements indicate different types of design knowledge:

"A stack is implemented by an array and a pointer to indicate the top element...."

"Quicksort is more efficient than Heap-sort if the elements to be sorted can be kept in main memory...."

"A modular system should have high cohesion within the components and low coupling between the components...."

The design knowledge involved in the prototyping approach will be concentrated on finding patterns in the conceptual model and translate these patterns to corresponding patterns in other modeling languages, or to statements in traditional programming languages.

Domain knowledge may also be formalized and integrated in a translation assistant. Whereas design knowledge is formalized to perform general translations, domain knowledge can be integrated to guide the translations. As such, the translation will be tailored to the specific domain. The issue of including the domain knowledge in translations has to weigh two conflicting concerns: flexibility and coverage.

If domain knowledge (e.g. formulas from physics, mathematics, economics, etc.) is included in the translation (the domain-dependent approach), powerful translations can be supported without external influence. However, the main drawback is lack of flexibility. The approach works perfectly within a specific domain, but has limited applicability outside the domain. Examples of domain-dependent translational system are ΦNIX ([12]) that generate the software that controls and records the data from oil well logging tools and WATSON ([76]) that aims at helping telephone engineers to achieve mathematically precise specifications of their domain.
In the domain-independent approach domain knowledge is not integrated in the translation system. The domain knowledge is input to the translation. This is a more flexible approach with high applicability. However, a major challenge of this approach is to minimize the number of output representations from an input representation. Thus, the translation has to be guided to obtain the correct output. The coverage and power of the translations are therefore reduced as compared to the domain-dependent approach. KBSA and its successor ARIES apply the domain-independent approach [73].

Thorough discussions on various aspects of translational approaches and presentation of translational systems can be found in [97, 117].

5.4.5 A Compiler View of the Translations

The principles for translating conceptual models to executable prototypes are similar program compilation. To provide the technical requirements for integrating a translation assistant within a CASE environment, we will use the general architecture of a compiler. The following discussion is very much based the one found in [88].

A general compiler architecture is shown in Figure 5.4. The architecture performs the compilation of an input representation (program) to an output representation (machine code) through six phases: lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator, code optimizer, and code generator. The purpose of the different phases will be briefly described below. Also, the error handler and the symbol table for managing the compilation are described.

![Figure 5.4: A general compiler architecture (from [88]).](image-url)

The lexical analyzer reads the input representation to identify its different elements. A lexical analyzer for a 3GL program reads the program from left to right to group sequences of characters that have a collective meaning into tokens.
The **syntax analyzer** organizes the input tokens into grammatical phrases that are used to synthesize the output. Grammatical phrases of a textual representations are often expressed in *parse trees*. A major task of the syntax analyzer is to ensure that the program does not violate the grammar of the language.

The **semantic analyzer** checks whether the source program contains any semantic errors. In traditional compilers, *type checking* is a major activity in this phase.

The **intermediate code generator** is used by some compilers to generate an intermediate representation that makes the final translation to the output representations easier. In some cases the intermediate representation is executable machine code. However, at this stage efficiency have not been taken into account.

The **code optimizer** attempts to improve the intermediate representation with respect to efficiency. A general approach for an optimizer is to reduce the number of instructions.

The **code generator** executes the actual translations by applying a *translation schema* which consist of a set of *translation rules*. The rules takes patterns of the intermediate representation and produce statements in the output representation. A translation rule can “graphically” be illustrated on the following form [23]:

\[
\frac{i}{o} \quad [c]
\]

Here, \(i\) is a pattern in (or a subset of) the intermediate (input) representation, \(o\) is a pattern in (or a subset of) the output representation, and \(c\) is an *enabling condition*, stating which type of pattern that should be triggered. The enabling condition is optional. Given these terms, a general translation algorithm focuses on *pattern matching* and can informally be stated as follows [117]:

```
begin
read the input representation and the translation schema;
repeat until end of input
   if a pattern in the input representation matches an i among the translation rules
      and the pattern is not violating the associating c then
         write o of the relevant translation rules into the output representation;
end of repeat;
end;
```
A major requirement to a translation schema is that it should be *meaning-preserving*. That is, the translation rules should preserve the semantics of the input representation when generating the output. In cases of generating executable programs, there should be a well-defined correspondence between the execution semantics of the source language and the target language.

Moreover, the translation schema should ensure *traceability* so that the input representation is kept trace of in the output representation. The translation schema should avoid, as much as possible, altering the structure of the source representation or introducing new variable or function names in the target representation. Thus, it should be easy to modify the source representation on the basis of evaluating the target representation.

**The symbol table** keeps track of the identified tokens and their various attributes. For a 3GL program these attributes may provide information about the storage allocated for its identifier, its type, its scope, and argument/passing methods for procedure and function calls. Generally, a symbol table is a data structure containing a record for each identifier, with fields for the different attributes.

Throughout the compilation phases, the information in the symbol table is used for different purposes. During the lexical analysis, the various tokens are detected and their corresponding identifiers are entered in the symbol table. Furthermore, in the semantic analysis, the type information is required.

**The error handler** reports on errors that are encountered during the compilation. According to Aho et.al a compiler should allow the compilation to proceed so that all errors in the source specification can be detected. Moreover, the detected errors should be reported in a readable fashion, explaining the place (e.g. code line) and the reason (e.g. "illegal symbol") for the error.

### 5.4.6 Towards a Tool Supported Prototyping Approach

The baseline of the prototyping approach has now been established. The approach can be related to conceptual modeling we obtain an overall architecture for the approach. This is shown in Figure 5.5. The four major activities involved are [93]:

**Modeling/Verification** During the modeling process, domain knowledge is captured and modeled. The modeling activity is driven by the underlying modeling language and the method for applying the modeling constructs. For instance, the DFD language recommends a top-down way of modeling a domain. Verification is usually provided an integral part of the modeling process.
Translation Translations may be used for two purposes:

- They may be integral parts of the modeling activity. If the employed languages have overlapping semantics, information from one model can be extracted and translated to new representation forms. Provided that the translation is correct, it can ensure that the various models are consistent with respect to each other. Also, translations will speed up the construction of new models.

- Translations may be used to generate executable prototypes from the dynamic parts of the model. For the translations to work, the model must be syntactically correct. This means that syntactical checks should be performed prior to the translations. As indicated in Figure 5.5, translations are established based on formalized design knowledge. It is envisaged that translations to executable prototypes should take place when critical parts of the model are difficult to comprehend only by inspecting the model.

Execution When an executable prototype has been established, it should be executed to assess the dynamic properties of the conceptual model. It is expected that the execution better will reveal the model behavior than model inspection would have done.

Validation The last task is validation. The experiences gained from executing the prototype should be compared with the expectations and intention of the users in the domain. If deficiencies or anomalies are detected, the conceptual model is revised and a new prototype is developed. As the modeling activity, active user participation is important for the validation to be successful, and should be encouraged.
The prototyping approach is inherently cyclic. The knowledge about problem domain and requirements to the information system are developed through iterative sequences of modeling, translation, execution, and validation. The user’s feedbacks from experiencing the behavior of these models guide the modifications introduced in subsequent model revisions. The cycle terminates when the behavior reflects the intentions of the user, and the model is deemed to be a satisfactory basis for the design and implementation of the information system.

As indicated in the figure, there are relationships between verification, translation, and validation:

- The purpose of verification is, in addition to assure the syntactic quality and consistency of the model, to prepare the model for translation. The syntactical and semantical checks aim at ensuring that the model consists of patterns defined in the translation rules. As such, translation plays an integral part of model verification.

- Translation is a major technique for improving model comprehensibility and simplifying the validation process. Also, by using translations to speed up the modeling process, it contributes to more effectively reach model completeness. The consistency of the model can be preserved as well.

In the next chapter the above architecture for model quality assurance will be included in a CASE tool context.
Chapter 6

Tool Support for the Prototyping Approach

The baseline for the prototyping approach has now been described. In this chapter we will establish the requirements for realizing the approach within a CASE tool.

The overall architecture for a CASE tool supporting translational prototyping is shown in Figure 6.1. The activities from Figure 5.5 are indicated. Moreover, the CASE tool should distinguish between the presentation and the representation of the model. The former denotes the model as it appears to the user of the tool, whereas the latter denotes the model as it is represented/stored within the tool.

The architecture mainly follows the principles of an ICASE tool and consists of four integrated components:

- A modeling environment which supports the user in constructing the model,
- a repository which allows for storing model representations,
- a translation assistant which translates a conceptual model to an executable prototype, and
- a prototyping environment which executes the prototype to reveal the dynamic properties of the model.

The requirements to the four components are discussed in Sections 6.1, 6.2, 6.3, and 6.4, respectively. Numerous requirements for each component can be established. In the sequel we will restrict ourselves to architectural requirements and emphasize the integration of the components.
6.1 Requirements to the Modeling Environment

An overall architecture of the modeling environment is shown in Figure 6.2. The following discussion is based on this architecture.

In the modeling environment, the developer is constructing the conceptual model. Assuming a visual conceptual modeling language, graphical facilities are a major prerequisite for supporting the model construction. Such facilities can be provided by workstations or personal computers. The user interaction is supported by a keyboard in combination with a mouse.

The modeling activity is supported by the drawing editors that are particularly tailored to the employed modeling language. A drawing editor should mechanize the manual drawing routines in ways that resemble “pen and paper-techniques”. Moreover, editing facilities like “create”, “delete”, “copy”, “move”, etc. should be supported.

A tool that integrates several drawing editors should have a common “look and feel” [143].
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 [147]:

- **Implicit checks**: This type of checks adapts the principles of *syntax-directed editors*. Thus, only modeling constructs that are defined in the language’s alphabet 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 and prevent the user from making *morphological errors*.

- **Explicit checks**: 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.

More advanced verification methods address the semantic quality of the model. In order to detect contradictory statements, consistency checks should be provided. To realize such checks, facilities that reason about the interpretation of the different modeling constructs should be embedded. A thorough discussion of consistency checks in CASE tools are found in [158].

The results of the different checks should be made available through an error handler, stating where in the model the error occurred, and, if possible, the reason for the error.

In addition to facilities for model construction and verification, the modeling environment should access the repository by means of storage facilities. These features should allow the user to "store" and to "load" representations of the models.

Printing facilities, report generators, etc. are also features that can be included in the modeling environment.

### 6.2 Requirements to the Repository

The repository provides storage for the model specification statements and consists of a storage medium and a storage schema. The storage medium may either be a database, files, whereas the storage schema defines a general format for structuring the model specifications. The schema can be directly be derived from meta model of the employed language.

If the repository is used to integrate several tool components, additional properties are required to manage the stored information as a consistent whole. This mainly concerns the exchangeability of data among the different tools. A common repository is expected to have properties like [143]:

- **interoperability**, e.g. that two tools can access each others' data,
- **non-redundancy**, e.g. little duplicate data, and little data that can be automatically derived from other data,
- **data consistency**, e.g. how well do two tools cooperate to maintain the semantic constraints on the data they manipulatable,
- **data exchange**, e.g. what must be done to make the data generated by one tool manipulate by the other, and
- **synchronization**, e.g. how well does a tool communicate changes it makes to the values of nonpersistent, common data so that the other tools it is cooperating with may synchronize their values for the data.
Figure 6.3: A general architecture of a translation assistant.

A thorough discussion of the construction of a CASE repository is found in [4].

6.3 Requirements to the Translation Assistant

The general architecture of the translation assistant is shown in Figure 6.3. The components and how they are related are based on the compiler architecture described in Section 5.4.5. However, the assistant has to integrated with the other components of the CASE tool.

Major parts of the lexical, syntax and semantic analysis are integrated in the verification support (syntax and consistency checks) of the modeling environment. This is indicated in Figure 6.3. Also, when errors are encountered during translation, the error handler should give meaningful messages to the user. Since the syntactic and semantic checks are included in the modeling environment, major parts of the error handler is part of this environment as well.

In the compiler architecture the symbol table defines the data structure for the input representation to the code generator. Thus, it resemble well the repository concept in a Case tool context.
The intermediate code generator and the code optimizer address the efficiency of the generated code. In a prototyping approach, these components are optional. This is indicated by "dashed boxes" in Figure 6.3.

The code generator is the main component of the translation assistant. It takes a syntactical correct model as input and generates the executable prototype according to the translation schema defined for the translation assistant. The translation rules are constructed based on design knowledge that matches syntactical patterns in the source and target languages. Thus, they are represented with basis in the formally defined syntax of the source and target language. Moreover, for the translation assistant to generate a correct executable program, the translation rules has to invoke the execution semantics of the source and target language. Subsequently, the knowledge of the correspondence in execution semantics of the language is included in the rules.

The quality of the translation assistant is dependent on the design knowledge that is formalized in the translation assistant and determine the quality of the generated prototype. Subsequently, the assistant plays a major role in determining whether the prototype — judged by the quality criteria in Section 2.3.3 — have the quality of evolving into the final information system or not.

For the purpose of improving model comprehensibility, though, efficient code is critical. The features of the prototyping environments is of more significance.

6.4 Requirements to the Prototyping Environment

An overall architecture for the prototyping environment is shown in Figure 6.4. By translating a conceptual model to an executable prototype in a specific target language, the properties of the prototyping environment is very much decided by the chosen target language and its environment. Also, the target language is a major concern for facilitating a smooth integration between the prototyping environment and the rest of the CASE tool.

In the sequel we will discuss the following areas of the prototyping environment: execution mode, test data, and presentation styles. The requirements are very much derived from [65].

Execution Mode

The execution mode for the prototype execution style can be classified in two main types step-by-step execution and batch execution.
Figure 6.4: An architecture for the prototyping environment.

**Step-by-step execution** lets the users communicate with the prototype during execution, giving inputs to the executing model. The prototype responds to the inputs according to its specified dynamics in a single step. The execution is controlled by breakpoints and spypoints similar to those found 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.

**Batch execution** The alternative to the interactive mode of operation, is to store or program test data and events on a separate file, and then run the prototype as batch jobs. This naturally limits users involvement to observation of outputs, but it may be useful if complex scenarios of the execution are to be set up. Programmed execution allows inputs to be represented as statistical distributions, as is customary when executing models that are written in simulation languages.
Values of Test Data

The input to the execution is artificial test data that resemble domain knowledge in the user’s environment. As indicated above, they may be given in an open, interactive fashion through a step-by-step execution or on files in connection with batch execution.

Basically, there are two kinds of values computed by an prototype that can be used in combination with the above execution modes: concrete and symbolic. The normal case is execution with concrete values.

Symbolic execution is a specific technique that makes use of symbols having value space in contrast to concrete values. The value space defines the constraints of the execution, but allows for a large subset of the possible behaviors to be examined at once. If the execution is performed on concrete values, a large number of test-cases is usually needed to explore the same value space.

Moreover, by symbolic execution all the computational detail of the prototype is usually not needed. This means that models that are not completely specified can be subject to prototyping.

Presentation Styles

While inputs normally are textual, outputs of an execution may be both textual and graphical. Textual outputs may be given as concrete or symbolic values, traces, and reports of various kinds. Graphical output is given to depict the current state of an execution.

The basic output of an execution is the execution trace. It contains the historical information of the execution as well as the registered (logged) data for each execution state. Based on the information contained in the trace, several presentation styles can be envisaged:

- **Textual**: The trace can be shown during the execution to report on the current state-of-affairs. Furthermore, the information can be collected at the end of the execution to make an execution report. Depending on the information available statistical analysis can be performed to produce execution statistics.

- **Graphical**: If there is a strong integration between the execution prototype and model, the result from the execution can be presented through the model elements. Animation is a graphical presentation style where the textual output of the executing prototype is fed back to the model to indicate the execution path. The technique assumes that by visualizing the execution it becomes more comprehensible.
Furthermore, if statistical information is produced, it can be accessed through the model. Provided that the information is related to model objects, it should be accessed by means of the objects. For instance, if the time consumption of process executions in a DFD model is available, it should be accessed by “clicking” on the process. Again this requires a strong integration between the modeling environment and the prototyping environment.

6.5 A Short Survey of Related Work

The ideas of prototyping and using translations to automate and support different development tasks have been known for several years, but is not widely used in practice. The approach suggested builds on many of the same principles for model transformations found in other environments. In contrast to domain-dependent transformation systems like bf WATSON ([76]) and \( \Phi \)NIX ([12]), our approach is domain-independent. That is, the conceptual model represents the domain knowledge independent of the transformation rules. As such, it basically follows the same approach as KBSA and its successor ARIES ([73]). Such an approach is more flexible and has a wider applicability than domain-dependent approaches.

Many research environments support prototyping by generating high-level program code from conceptual models. The target languages of translations vary considerably in different environments. For instance, in TEMPORA ([138]), PROLOG is generated, whereas C and Ada is produced in STATEMATE ([66]). Also, in STATEMATE ([66]) a prototype in a simulation language is generated.

Of the different execution modes, step-by-step execution is found in [15, 66]. Batch execution is also found in STATEMATE ([66]). Symbolic execution is well explored in the the GIST environment ([11, 31]), but is also found in environments that are built around Petri-nets [98]. Executing models in these languages is symbolic, in that the existence of a token in a place is an abstraction of the presence of a computed value. In Teamwork/ES ([15]), a similar approach is taken, where models written in a Transformation Schema ([151]) are executed symbolically. The different kinds of flows in a models can hold tokens during execution, this being a representation of the fact that the flow has some value.

In the remaining of the thesis, we will report on how the prototyping approach is realized in the PPP environment.
Chapter 7

The PPP Environment — Realizing the Prototyping Approach

An architecture for the prototyping approach has now been established. In the following chapters, we described how the approach has been realized in the PPP (Phenomena, Processes, and Programs) environment. The presentation will illuminate both the PPP conceptual modeling languages and the tool support for the approach.

In this chapter we will establish the basis for the presentation and evaluation of the PPP environment. In Section 7.1 the PPP modeling approach is outlined. In order to illustrate the features of the PPP language and the PPP tool, a description of a bank domain is provided in Section 7.2. Throughout the presentation of the PPP environment, different aspects of the environment will be evaluated. Sources that support the evaluation are introduced in Section 7.3.

7.1 The PPP Modeling Approach

The PPP language provide mechanisms for the PPP modeling approach. The language is an integration of four basically independent sub-languages, each having its own diagrammatic notation. The sub-languages are still grounded on widely accepted modeling languages and address different aspects of a problem domain and its information system:
- The **PrM (Process Model) language** is used to describe functional structures of a domain. These structures show the decomposition of systems into subsystems, the interactions between subsystems, and the translation of data in a particular system or subsystem. They document the functional architecture of the resulting information system. That is, the structures explain the functions of a system, and describe how these functions are constructed by combining other functions. The PrM language is based on the traditional DFD, but aims at eliminating some of DFD's shortcomings discussed in Section 4.1.2. Thus, it has added constructs for modeling formal domain knowledge. PrM has some characteristics in common with Translation Schema [151], and other languages like [81].

- The **PhM (Phenomenon Model) language** is used to model static structures in the domain. The PhM language is an extension of the ER language, and includes many features of newer structure-oriented modeling languages [114].

- The **PLD (Process Life Description) language** is used to specify process logic of bottom-level processes. The structure of such processes is regarded sufficiently simple (atomic), and can be expressed by an algorithmic description. The PLD language has many similarities with block-structured, program design languages [144].

- The **UID (User Interface Description) language** is used to describe static and dynamic aspects of user interfaces, based on the possibilities offered by graphical user interface technology.

The PPP modeling approach guides how the languages shall be used to model the domain. Although no firm guidelines has been established, the PPP modeling approach mainly follows a top-down strategy.

The early phases of the modeling approach is devoted to problem domain understanding. The assumption is that knowledge and functional requirements are fuzzy early in the modeling process process and should be represented as such. The PrM and PhM languages represent the "conceptual part" of the PPP language and should therefore be used in the very early stages of problem understanding and formulation. As the requirements/knowledge become increasingly more concrete, the languages have mechanisms for representing such requirements. Hierarchical decomposition in PrM is one important mechanism for this purpose. Such an approach will also encourage depth-first analysis. Requirements/domain knowledge that are regarded as critical for the future system can be isolated in the model. This part can then be analyzed in detail, while other parts of the model are overlooked.

The conceptual model of the system and its environment is decomposed until it is possible to make the decision on what parts of the real system that is going to be computerized. A definition of this automation border is important since it addresses the question of interfaces to the computer system. To describe the interface precisely, the UID language is used. The functional analysis continues for the future
7.2 Domain Description of a "Simple" Bank

computerized part. The decomposition stops when each low level process can be described through the PLD language.

A main idea behind the PPP modeling approach is to capitalize on existing information available in the constructed models. Since the four sub-languages have a partly overlapping semantics, information can be shared among the different sub-models. This supports the integration philosophy of the PPP language and will be further discussed throughout the presentation of the PPP environment.

7.2 Domain Description of a "Simple" Bank

To illustrate the feature of the PPP language and the PPP tool, we will use an example domain description of a "simple" bank. A complete documentation of domain description and how it has been modeled by the PPP language is found in [59]. The domain is described as follows:

Our domain includes some of the more simple tasks in banking systems. Focusing on the processing of transactions and loan applications, we have the concepts of customer, account, loan, payments, transaction, and loan application. These are related to each other in various ways.

A customer can have several accounts and loans registered. The customer may deposit or withdraw money from his account by sending a transaction to the system. If the transaction is valid, i.e. the given account exists and the given name is identical to the account's owner, the system will update the account and afterwards issue an account statement to the customer. A withdrawal is only accepted if it does not lead to a negative balance.

Every month account statements are sent to the customers, specifying all the transactions made the last month. This is a completely automated process, and the system itself checks the accounts' balances and associated transactions, and produces the statements to be used.

Applying for a loan, the customer must first have opened an account in the bank and must have a good credibility. Moreover, the balance of the account must never be negative. 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 accepts the loan offered, the customer is invited to sign a contract which specify among other things an interest rate and a payment plan.
7.3 Some Remarks on the Evaluation

Throughout the presentation of the PPP environment, different aspects of the environment will be evaluated. The properties of the PPP language will be discussed as well as the tool support. The evaluation is meant to form a basis for future improvements and extensions of the PPP environment.

The evaluation is very much based on the arguments and examples documented in this thesis. Furthermore, throughout the development of the PPP environment, several case studies have been carried out. Results from the following studies will supplement the evaluation:

**Modeling exercises** have used the PPP Language to model different domains. The most comprehensive exercises are the *Library Case Study* ([96]) and the *Office Information Case Study* ([102]). The former study aimed at modeling the dynamic part of the library services at the University of Trondheim, BIBSYS, where as latter study modeled our department as an example of an office domain. Both studies had a reasonable size and contributed well to the understanding of the properties of the PPP language. Moreover, the Library study were supported by the PPP tool. Valuable feedback to the functionality of the tool was therefore provided during the study.

**Comparative studies** have been carried out to some extent. Some properties of the PPP Language has been evaluated and compared with other modeling languages. In [7] the expressiveness of the PrM language with respect to three different domain types is evaluated: (1) a transaction-oriented domain, (2) a real-time domain, and (3) decision-support domain.

An extension of this study was used as a student exercise. The PrM language and the DFD language were used to model descriptions from the three domains. Thus, the language could be evaluated and compared relative to the three domain. Along with this student exercise, the PPP tool was used to support the modeling process. Thus, the exercise gave valuable feedback to the functionality of the tool as well.

Finally, internal reports from and discussion with persons who have been involved in the development of the PPP environment will be used to illuminate different aspects of the environment.
Chapter 8

The PPP Language

This chapter presents the PPP language. The presentation very much builds on similar presentations of the PPP language found in [58, 88, 89, 155]. The four sub-languages of the PPP language are presented in Sections 8.1 — 8.4, respectively. The presentation will be informal and the basic modeling constructs of each sub-language are explained with reference to our bank example. Most emphasis will be put on the PrM and PLD languages due to their executable properties. As such, they will form the basis for PPP models that are translated to executable prototypes. In Section 8.5 we explain how the sub-languages are integrated. The properties of the PPP language are discussed in Section 8.6.

8.1 The PhM Language

The PhM Language [111, 130] is used to model the static aspects of the problem domain and the static objects of the information system. Since the PhM language exploits the simplicity and elegance of ER diagrams, the general structures are the same. However, to improve the expressiveness of the language, several extensions have been made. Some of them are shown in Figure 8.1 where parts of the static aspects of our bank example are modeled. PhM constructs are labeled. Compared to the original ER language three new constructs are shown: attribute relations, coverage of relationships, and subclass relationships.

An attribute is a relationship between an entityclass/relationshipclass and a data type. There are four different types of attribute relations. An entityclass must have an identifier attribute, which uniquely identify an instance of the class. An identifier is indicated by the abbreviation 'id', as for account.id for entityclass account. A repeating group relates an entityclass to a set of values of the same data type. A quality is an attribute of the entityclass as a whole, for instance average.balance of account. Any other properties are simply called attributes. We have omitted
Figure 8.1: Parts of a PhM model for the bank system.

detailed modeling of attributes to simplify the example.

In addition to the ordinary cardinality specification found in the EER ([139]), the PhM language provides mechanisms for specifying the coverage of a relationship. The coverage of a relationship may be full, meaning that every instance of an entityclass must participate in the relationship, or partial, meaning that participation is only permitted. In the example, we see that several transactions (N) can be processed on one account (1). Moreover, each transaction (f) must be related to an account, but there may exist accounts (p) on which no transactions yet have been processed.

A subclass represents a subset of an entityclass, of a relationshipclass, or of another subclass. It can have only one superclass, and it inherits the identifier of this superclass. An instance of a subclass is always member of the set of instances of its superclass. In the bank model, no subclasses are modeled.

For a more comprehensive description of the PhM language, the interested reader is referred to [110, 130].

8.2 The PrM Language

The PrM language is used to model functional aspects of a system and the interaction with its environment. A top level PrM model for the bank example is shown in Figure 8.2 and modeling constructs that differ from the traditional DFD language are labeled.

Processes have the same meaning here as in a DFD, i.e. describing the business activities as translation of input flows to output flows. From our domain descrip-
tion, four activities for transaction processing, loan application processing, account opening, and issuing of monthly balance statements can be identified and modeled as processes. Processes can be decomposed in the same manner as done in the DFD language. Process P1:Process transaction is boldly marked in Figure 8.2 to indicate that it is decomposed and that the detailed specifications of the process are revealed in Figure 8.3. The decomposition consists of five subprocesses, P1.1 through P1.5.

**Flows** and **stores** also have the same meaning as in a DFD. However, the PrM language allows flow contents to be specified as variables or attributes from the PhM model, with accompanying type information. A set of variables with type information is called **items. External agents** have the same semantics as external entities in DFD. The name difference has been made to emphasize the dynamic aspects of entities. In the top level model, the **Customer** is the only external agent.
Timers increase the temporal expressiveness of PrM. In the example, a timer is used as a clock which send output signals on a monthly basis to initiate issuing of balances. Input signals may start and stop timers. They can also be used as delays, where output signals are sent some specified interval after an input signal has been received. Generally, input signals may start and stop timers. In Figure 8.2, a timer (of clock type) is included to regularly send Statement_schedule each first day of the month.

Control flow is modeled by the use of triggering and terminating properties of data flows. These properties determine when a process will start and stop its execution, respectively. If a process is passive and receives the right combination of triggering flows, it will start executing. On the other hand, an active process terminates by sending a combination of terminating flows. Non-triggering and non-terminating flows can be sent/received any time while the process is active. In the example, process P3 can be activated if it receives the triggering input flow Customer\(^1\), and will terminate by sending the terminating flow Account.

To define logical relationships between input flows and output flows, PrM offers input ports and output ports, respectively. There are three basic kinds of ports corresponding to the three logical connectives: conjunction (AND), disjunction (OR), and exclusive disjunction (XOR). From the example model in Figure 8.3, we see that the input port of P1.1 is an AND port with two input flows, i.e. all input flows must be received during an execution of the process. The outer output port of the process is an XOR port, meaning that only one of the flows are sent in

---

\(^1\)Marked with a 'T'.
order to terminate the execution.

Moreover, a port may be *conditional*, *repeating*, or both in any combination. A conditional port reflects a situation where flows are sent or received depending on some condition. A repeating port means that flows may be received or sent a number of times during a single execution of a process. Also, *composite ports* can be formed by placing ports inside each other. For instance, the input port of P1.2 corresponds to the logical expression:

\[
\text{AND(Withdrawal\_transaction,COND(New\_withdrawal))}.
\]

The dotted line of the AND port for the input flow *New\_withdrawal* indicates conditionality\(^2\).

Now, we can describe a single execution of process P1.2 as follows: If the process is passive, it can be activated by receiving the flow Withdrawal\_transaction. It terminates by sending either the flow Aborted\_transaction or (exclusively) the flow Ok\_withdrawal. During execution, it may receive the flow *New\_withdrawal*, and it may send the flow Withdrawal\_rejection.

For more detailed descriptions of PrM, see [58, 111].

### 8.3 The PLD Language

The PLD language is used to specify process logic of bottom-level processes. The processes that have associated PLD models will be denoted *automatic processes*, whereas processes that emulate human tasks will be called *manual processes*. Constructs for assignments, iteration, and choices are defined. In addition, two constructs for receiving and sending data provide interprocess communication and communication with users and databases.

Figure 8.4 shows the process interface of process P1.2 reflected in a PLD model. The different modeling constructs of the PLD language is labeled. The initial PLD model is automatically generated from the PrM model in Figure 8.3. The items are inherited in the PLD models. Also, the ports and triggering/terminating flow properties define the overall structure and the ordering of the PLD model, respectively. The translation algorithm will be further explained in Section 9.3.11.

To complete the model the developer has to manually fill in the lacking PLD statements. This statements are shaded in the PLD model in Figure 8.4.

The control flow of a PLD model is top-down and from left to right. The *start construct* simply marks the start of the PLD model. First, the process receives With-

\(^2\) Repetition is indicated by an unbroken line.
drawal_transaction. The receive construct holds this information, together with variable names and types of the data received. Here, we see that the transaction information includes account number, account balance, customer name, amount, and date.

A choice construct follows, to distinguish between two alternatives; either the balance exceeds the specified amount, or it does not. The choice construct is a compound of one selection construct which marks the choice situation, and two or more alternative constructs, one for each alternative to be evaluated. Each alternative contains an expression. If this expression evaluates to true, the block below the alternative construct will be executed.

For the case that the balance is not OK (account_bal < amount), the available amount (Withdrawal.rejection) is sent to the customer, with a request for a new amount. This amount is then received (New_withdrawal) and if it still not OK, an aborted transaction note Aborted_transaction is sent to the customer. On the other hand, if the transaction is OK, OK_withdrawal is sent to P1.3 in order to update the account. The send construct identifies the data flow and the receiver. In addition, it contains the item information that is sent.
8.4 The UID Language

The assignment construct is depicted as a simple rectangle containing the variable to be updated and an update-expression. The loop construct is not illustrated in the example model. It construct contains an expression to be evaluated each time the loop body can start execution, and corresponds to a WHILE-loop or a FOR-loop of high level programming languages.

8.4 The UID Language

The UID language is used to model details of the human computer interaction part of an information system — the user interface. An UID model will describe how to use the system, how to exchange information between user and system, and when to activate the various system functions. The UID language has been adapted to windowing technology and is divided into two sub-languages: The User Interface Presentation (UIP) language and the User Interface Dialogue Description (UIDD) language.

The UIP language is used to model the presentational or the static part of the user interface. Thus, a UIP model consists of objects like windows, menus, input/output fields, etc. The objects may be atomic or compound and their properties determine appearance, functionality and relationships to other objects. The notation for the UIP language is derived from the PhM notation.

The UIDD language is used to model the dialogue or the dynamic part of a user interface. A UIDD model connects all objects defined in the UIP model, calls application procedures, and processes user interface data. Each callback procedure which is defined in a UIP model, is described by the UIDD language and one UIDD model corresponds to only one callback procedure. The UIDD notation is in fact identical to the PLD notation. However, the semantics of the send and receive constructs are slightly changed. In the context of the PLD language, these concepts are used to describe communication or exchange of data between PLD model. Within the UIDD context, they are used to model procedure and function calls.

The UID language will not be further exploited in the remaining of the thesis. For the interested reader, a thorough presentation of the UID language is given in [74].

8.5 The Integrated PPP Language

The PPP language is an integration of the four sub-languages presented above. The links between the sub-languages are shown in Figure 8.5 using the different submodels from our bank examples. The integration is summarized as follows:
The PhM model is used to specify the contents of PrM flows. Every flow in the PrM model must be characterized by means of the information it provides. Since that information is reflected in the PhM model, the flow will be associated with certain entities, subclasses and/or relations in the model. Accordingly, the PhM model specifies data to be processed and translated in the system. The process of defining the content of PrM flows using the PhM model will be further explained in Section 9.3.6.

The bottom-level processes of a PrM model is characterized by a simple and sequential behavior. To specify that simple behavior, the PLD language is used. An initial PLD model is created based on the structure of the input and output ports and on the triggering and terminating properties of the input and output flows. The process of linking PrM and PLD models will be further explained in Section 9.3.11.

So far, the UID language has not been formally integrated with the remaining PPP languages. Different integration strategies are proposed in [74], though. One of the strategies is indicated in Figure 8.5 and its main features are summarized as follows: A UIP model is created based on the content defined for flows between agents and processes. Subsequently, the UIP model is implicitly linked to entities and relationships in the PhM model. Furthermore, the UIDD model is created based on the UIP model and PLD model to the bottom-level processes that describe user-system communication.
8.6 Properties of the PPP Language/Model

In the following we will discuss the following properties of the PPP language that are relevant for quality assurance of PPP models: syntax, expressiveness, and comprehensibility.

8.6.1 Syntax

Although not documented in this chapter, the syntax of the PPP language has been formally defined. The syntax for the PrM and the PhM languages has been documented in [111] using first order logic. Moreover, syntax for the PLD language is represented in a BNF-notation in [153].

8.6.2 Expressiveness

Since the PPP language integrates four sub-languages, each addressing different aspects of a domain and the resulting information system, PPP a highly expressive modeling language.

The executability property is preserved in the dynamic parts of the language: the PrM language and the PLD language. Important, these languages have control structures so that different execution patterns can be specified. The triggering/terminating flow properties and the process ports can be used to model dynamic properties like precedence, exclusiveness, concurrency, and synchronization, etc. In [84] this is thoroughly documented and an algorithm for translating PrM models to Petri-Net models is proposed.

Although the PPP language in this chapter has been used to model a bank domain, the case studies have shown that the PPP language is suitable for modeling a wide range of problem domains. The language semantics will correspond better to some domains than others, though. For instance, the library case study showed that the language was particularly suited for modeling transaction-oriented domains [96]. On the other hand, the PPP language provides no constructs for explicitly model roles and responsibilities within an organization [102].

The PPP language is developed to model functional domain knowledge/requirements. In [112] and in [62], the PrM language is extended with probability distributions. As such semi-deterministic properties of the problem domain can be modeled. In [112] Opdahl shows how the PrM language can be particularly tailored towards performance modeling.
8.6.3 Comprehensibility

Since the PPP language is based on well known languages like DFD and ER, it inherits the simple and intuitive nature of these language. Emphasizing the familiarity factor, PPP should be easy to learn and use for people acquainted with these languages. This view is in general supported by the case studies mentioned in Section 7.3.

However, the introduction of a triggering/termination feature and in particularly the port construct may hamper the comprehensibility. [102] claimed that the port concepts could be hard to catch for non-experienced developers and hampered the communication.

A major weakness of the PPP language is its lack of uniformity [129]. Particularly, the hierarchical properties of the language is restricted to functional decomposition in the PrM language and subclasses in the PhM language.

A general and uniform framework for hierarchical modeling, HICONS, is proposed by Sindre in [129]. Here, all aspects of a domain is treated uniformly with respect to their hierarchical properties. It consists of four standard hierarchical constructs: classification, aggregation, generalization, and association. Also, a fifth constructs, vague composition is defined to indicate that the structure of knowledge elements is undefined or fuzzy. The notation follows a onion-style which according to [129] is preferable to edge-style notation with respect to perceptibility.

To improve the uniformity of the PPP language without sacrificing its expressiveness, HICONS is an interesting alternative.
Chapter 9

The PPP Tool — Modeling and Verification

This chapter presents the PPP tool. It is built as a prototype version of a CASE tools according to the architecture presented in Chapter 6. The PPP tool is supporting the PPP language and has been used to investigate the feasibility of various aspects of conceptual modeling and quality assurance of PPP models.

The overall architecture of the PPP tool is shown in Figure 9.1. The hardware platform for the tool is SUN work stations. Unix and Sunview have been the basic system software during tool development. Moreover, BIM-Prolog has been the implementation language for the functional features, whereas PCE integrated with Prolog has been used for developing the user interfaces.

Drawing editors for the PrM, PhM, and PLD languages have been implemented. A drawing editor for the UID language is specified in [74], but has not yet been implemented. Verification support of the PPP model is provided as an integral part of the drawing editors. Validation support is so far concentrated around translating PPP models to executable prototypes. This is further presented in Chapter 10.

The basic characteristics of the repository and the user interface are described in Sections 9.1 and 9.2, respectively. The modeling environment of the PPP tool will be presented in Section 9.3 along with a modeling session. Finally, different aspects of the PPP tools are evaluated in Section 9.4.
Figure 9.1: The overall architecture of the PPP tool.

9.1 Repository

9.1.1 Storage Medium

In a multi-user environment the storage mechanisms of the PPP tool should be provided by a database management system. As the functionality of the modeling support and the quality assurance of PPP models were emphasized, the requirements to the storage medium were relaxed. So, the PPP models are stored as ASCII files under Unix. In the early versions of the tool a complete model was stored on a single file. To accommodate version control to a multi-user environment, newer versions of the tool use a multi-file strategy. Here, different parts of the models are stored on different files. For instance, a PrM model is stored in a file-hierarchy corresponding to the decomposition structure.

A further discussion of a storage strategy with respect to configuration management is carried out in [4].
9.1. Repository

Figure 9.2: Parts of the meta model for the PrM language.

9.1.2 Storage Schema

The storage schema for a PPP model is defined based on the syntax — or the meta-model — of the PPP language. Parts of the meta model for PrM and PhM are shown in Figure 9.2. The basic modeling constructs and their relationships are indicated. Also, the interpretation of the relationships is given. For instance, a PhM construct can be linked to many PhM constructs. Also, we see that the meta models are intergrated by the item construct.

In the PPP tool, the internal format of the storage schema is mainly influenced by the implementation language for the tool. To avoid extensive mappings between different formats, the meta model is internally realised in a prolog fact schema with the following general format:

\[ \text{ppp-construct}(\text{Id}, \text{Att}_1, \ldots, \text{Att}_n). \]
So, a PPP model is stored as a set of prolog facts.

In Figure 9.3 we have illustrated how parts of a PrM model are stored. Here, four different PrM construct (process, flow, import, and output) are represented as Prolog facts. Each object in the PrM model corresponds to one fact. Composite objects are compactly represented by exploiting Prolog’s list mechanism [29].

Input flows to and output flows from process P1.1 are represented as two lists containing the flow identifiers. The output port of process P1.1 is composite and corresponds to the logical expression XOR(f1.3, XOR(f1.4,f1.5)). As indicated in Figure 9.3b, the port is stored as two Prolog facts with identifiers op1.1 and op1.2. The first fact represents the inner port, whereas the second fact the outer port. Thus, op1.1 is contained in op1.2’s list of flows and ports.

It should be noted that the Prolog facts illustrated in Figure 9.3 is somewhat simplified. We have concentrated on showing how the model structure and the model content is stored. Layout aspects like the objects’ placement on the screen, and administrative aspects like breaking of flows, etc. are omitted. For the interested reader, a complete description of the storage structure for a PPP model is given in [157].
9.2 The User Interface and PCE

The user interface of the PPP tool has exploited the graphical facilities of Sun workstations. The window system Sunview has been the environment for the developers access to the PPP tool. To implement the user interface routines, we have used PCE integrated with BIM-Prolog. In the following, we will describe the basic features of PCE and explain some aspects of the user interaction.

PCE is developed at the University of Amsterdam (SWI) and facilitates quickly creation of user interfaces [5]. PCE has an object-oriented nature and provides a wide range of predefined classes for developing a graphic user interface. The main categories of object classes are: (1) windows which provide access to Sunview, (2) user interface allowing the user to communicate with application using various kinds of menus, buttons, keyboard accesses, etc., and (3) graphics providing both primitive graphics like lines, boxes, circles, etc. and compound graphical objects defined by the application. A specific user interface is therefore composed of a set of objects from these classes.

The programming interface between PCE and Prolog is based on message passing and unification. PCE provides three types of actions to build the user interface that can be accessed from a Prolog application [5]:

Create objects builds a unique graphical object by the following PCE command:

\[
\text{new}(\text{Object},\text{Object-description}).
\]

Object is created by instantiating a predefined PCE class that is defined in Object-description. For instance, a window with name “PrM editor” is built by the following command:

\[
\text{new}(@\text{window},\text{window}('\text{PrM editor}'))
\]

Manipulate objects controls the structure and behavior of a created object by sending messages of the following form:

\[
\text{send}(\text{Object},\text{Behavior},\text{Value}).
\]

Object is manipulated by receiving a message of type Behavior with value Value. For instance, the size of the window created above can be set to 200*100 points by sending the following message:

\[
\text{send}(@\text{window},\text{size},\text{size}(200,100)).
\]

Access objects retrieves information from a created object. The structure or behavior is accessed by the following command:

---

1 Object references must be prefixed with @.
Figure 9.4: The effect of creating window using three PCE commands.

\[ \text{get(Object, Behavior, Value)} \]

Object, Behavior, and Values have the same meaning as above. For instance, the size of the window can be retrieved in the following command:

\[ \text{get(@window, size, Value)} \]

The predicate returns with \( \text{Value} = \text{size}(200,100) \)

In Figure 9.4 we have shown the effect of creating the window\(^2\). The window is opened at a particular point on the screen, \((70,50)\), by the third command.

The above syntax and the wide range of built-in classes make PCE a flexible and powerful tool for creating the user interface of the PPP tool. For a more thorough description of PCE, the reader is referred to [5].

### 9.3 The Modeling Environment of the PPP Tool

In the sequel we will present the modeling environment of the PPP tool. The architecture of the environment is similar to one discussed in Section 6.1. We will present the environment as the user sees it and carry out a modeling session in order to illustrate its layout and functionality. A PPP model of the bank domain is developed from scratch supported by the modeling environment. The session terminates when the PPP model is sufficiently complete to be translated to an executable prototypes. Both the modeling and verification support of the PPP tool are explained throughout the modeling session.

It should be noticed that the figures that illustrates the modeling scenario are not screen-dumps of the PPP tool. They are handmade to improve the perceptibility of

\(^2\)The proportions among the different sizes are not necessarily correct.
the tool's functionality. Also, explanatory comments can easily be attached to the figures. Although they basically resemble the tool as the user sees it, some figures are slightly “polished”.

Before starting the modeling session, it should be mentioned that the user interaction is carried out through mouse handling and menu handling. The different three buttons of the mouse attached to the Sun work station provide different functions. For instance, the right button is used for invoking menus, whereas the middle button is used for moving windows and objects around on the screen.

Moreover, to distinguish between different modes for the drawing editor, we use different cursors. For instance, a cross cursor, \(\varnothing\), appears on the screen when the cursor is in the drawing area, whereas an hourglass cursor, \(\varpi\), appears on the screen when the tool is working.

For the interested reader, a comprehensive user guide for the PPP tool is found in [71].
9.3.1 Creating the PPP Model

A modeling environment of the PPP tool is invoked from Sunview. The main window of the tool is shown in Figure 9.5 and consists of three subwindows: PPP command window, PPP overview window, and PPP editor window. Below, these are briefly described.

PPP command window

This window holds the basic functionality of the PPP tool at the model level. The user can choose among the following options: New (creating a model), Edit (editing an existing model), Print (printing a model), Copy (copy a model), and Delete (delete a model).

In Figure 9.5 we have shown the effect of selecting an existing model for modification. A window of available models appears on the screen, among which the developer selects the intended one; here bank. When the selected model is confirmed (clicking on the OK button), bank.ppp occurs in the command window as the current model.

PPP overview window

This window gives structural view of the “current” PPP model. In Figure 9.5 such a view of the bank model is shown. By providing a structural view of the model, the user may enter it at any level.

PPP editor window

This window contains the drawing editors provided by the PPP tool. Three editors — PrM editor, PhM editor, PLD editor — have been implemented. In accordance with principles for tool presentation integration, these editors have a common ‘look and touch’. When a model is built from scratch, one of these editors is invoked. In principle, the developer can invoke any editor. However, according to the PPP modeling approach, it is advisable to start the development with the PrM editor or the PhM editor.

It should be noticed that prior to selecting the appropriate editor a model has to be created. This is done by selecting the new option in the command window and by giving the model an appropriate name. In our modeling session bank is suitable. Technically, by creating a new model, a sub-directory in the Unix file system is built. In this directory all model files are stored according to the schema in Section 9.1.2.
Below, we will describe the basic functionality of the drawing editors by using the PrM drawing editor. Since the modeling and verification support of the PhM editor are uniform to the support provided by the PrM editor, we omit a thorough description of the functionality of the PhM tool.

### 9.3.2 Building the PrM Model

The PrM drawing editor is shown in Figure 9.6. Like the main window of the PPP tool, it consists of three main components as follows:

**PrM command window** provides the main functions for manipulating a PrM model. The functionality is accessed through pull-down menus that provide commands like save (saving a PrM model), print (printing a PrM model), etc.

**PrM drawing area** is used to construct the PrM model. The user interaction is provided by *mouse* for drawing actions, and keyboard for textual descriptions for model objects.

**PrM construct menu** contains images of the modeling constructs provided by the PrM language. Also, a symbol for adding plain text to a PrM model is given. By providing only predefined symbols for the user, the tool prevents the user for making morphological errors in the model.
In Figure 9.6 we have shown the PrM model #1 for our bank example. Also, the tool supports automatic numbering of the developed objects so that each object is uniquely registered.

To each PPP object that is created a popup menu is attached. The menu is context sensitive and contains only options that are relevant for the actual PrM construct. In Figure 9.6 the available options for the process construct are shown. Most options will be explained throughout the modeling session.

Figure 9.7: Syntactical checks of PrM model version #2.

In Figure 9.7 some names and flows have been attached. If the user draws a flow between an agents and a store, the grammar of the PrM language is violated and an error message occurs stating that this flow is illegal. The flow is automatically removed. Thus, it is implicitly supported by the tool.

### 9.3.3 Checking Syntactical Completeness

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

In Figure 9.8 version #3 of our evolving PrM model is shown. At this stage the model is still syntactically incomplete. For instance, process P1 is lacking output flows, whereas agent A1 is lacking a real name. By checking the syntactical completeness of the model, the tool will respond with error messages shown in the figure.

As indicated in Figure 9.8, 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.
9.3.4 Checking Unique Names

Some mechanisms for checking semantics quality of a PPP model has been implemented. From above we know that the tool ensures that the created objects are automatically given a unique identifier. One simple check concerns the usage of *unique* names within a model. In the current version, the user is warned if two or more PrM objects have identical names.

9.3.5 Decomposing the PrM Model

Assuming that the top level model is completed and the overall functional properties of the domain are specified. To describe the activities of the bank example in more detail, the PrM language allows for decomposition of processes. In lines with the main PPP philosophy of capitalizing on existing model specifications, information from the top level PrM model is inherited to the decomposed level. Input flows and output flows are automatically inherited. Identification of subprocess are automatically numbered, prefixed with the identifier of the mother process.

Having decomposed a process, the modeling process for this process can continue in the similar way as for the top level model explained above. In the sequel, we will present how flow content, triggering/terminating flow, and ports can be defined to further expand the PrM (PPP) model.

9.3.6 Defining the Flow Content

In the PPP modeling approach, a PrM model and a PhM model are linked together by defining the flow content — items — from the specifications of the PhM model. A major prerequisite is that the PhM model has been developed and that attributes
and data types have been attached to the entity classes and relationship classes.

In Figure 9.9 we have shown how the content of flow transaction is defined. The relevant part of the PhM model is depicted and flow content is defined by selecting entities/relationships or by picking attributes from different entities. In the former case all associated attributes will be inherited along with the entity/relationships. In the latter case, shown in Figure 9.9, “owners” of the attributes are recorded as well. The final flow content is shown in window Flow content.

It should be noticed that the content of stores can be derived from the flow contents. A store should contain the union of items that are defined for flows related to the store.

### 9.3.7 Defining Triggering/Terminating Flows

Input flows and output flows may have triggering and terminating properties, respectively. Definition of triggering and terminating flows is repeatedly carried out for each process. The input and output flows attaching the actual process are shown in windows. Here, the developer can easily select the triggering/terminating flows.

In Figure 9.10 we define triggering/terminating flows related to process P1.1: Verify transaction. The relevant input and output flows are shown in the two windows, respectively. The selected flows are marked as bullets in the windows and with “T”-s in the model.

From the definitions shown in the figure we can state that, on the input side, transaction will trigger the process, whereas previous balance that has no trigger-
Figure 9.10: Defining triggering and terminating flows.

9.3.8 Defining Input/Output Ports

Definition of input ports and output ports follows in many ways the same strategy as for the definition of triggering/terminating flows. The relevant process is selected and the attaching input and output flows occur in two windows, respectively. Flows that belong to the same port are selected from the window and the appropriate port type is chosen. Composite ports are recursively defined. The “inner” ports are defined first and can later be grouped together with other ports or flows.

Input and output ports to processes are highly related to the triggering/terminating flow properties. In [111] several rules stating how port can be constructed relative to triggering and terminating flow properties. For instance, all flows contained in an input/output XOR port should be triggering/terminating.

Figure 9.11: Defining input and output ports.

In Figure 9.11 we define input and output ports for process P1.1. On the input side, an AND-port contains the flows transaction and previous_balance. Thus, it corresponds to the logical relation AND(transaction, previous_balance). This means that information from both flows are consumed by the process during execution. On
the output side, a composite XOR-port is created. The inner output port is defined as XOR and contains `withdraw_transaction` and `deposit_transaction`. The port corresponds to the logical expression `XOR(withdraw_transaction, deposit_transaction)`. This means that the information on one of these flows is produced during process execution depending on the process logic.

### 9.3.9 Extensive Completeness Checks

The syntactical completeness checks explained in Section 9.3.3 were restricted to construct names and flow completeness. As the model has become increasingly complex and complete, more extensive checks can be initiated. Checks for the syntactical completeness of flow content, triggering/terminating flows, and input/output ports are provided by the PPP tool. They are all activated by the user. As indicated in Figure 9.8 scope of such checks is decided by user of the tool.

### 9.3.10 Checking Hierarchical Consistency

Ports can be interpreted as logical expressions. In this way, the consistency between different parts of the model can be checked. An algorithm that checks the consistency between a process port and the constructed port from the ports attached to the subprocesses is proposed in [128]. This algorithm has been implemented within the PPP tool [8]. A proposal for extending the algorithm in order to check consistency within “flat” PrM models is found in [158].

### 9.3.11 From PrM Models to PLD Models

When the processes have been sufficiently decomposed and the associated ports triggering/terminating properties flows have been defined, the PLD language can be used to describe the logic of bottom-level processes. Since the PLD language is used to describe automatic processes, a set of PLD models will define the automation border between the computerized and manual system. Those bottom-level processes that have no PLD model, is supposed to remain manual.

No formal rules for switching from PrM modeling to PLD modeling has been established. The following guidelines are given in [58], though:

- The process in question forms one semantic unit.
- There is no concurrency inside the process.
- No manual tasks are performed inside the process.
- It is straightforward to describe the process in a simple algorithm.
- If the process ports are very complex, it may be an indication that the process should be decomposed.

- Most processes at the lowest level will have three separate phases in their PLD: an input phase followed by some computation and an output phase. If inputs and outputs are mixed in large numbers, the process should probably be decomposed.

1. Input flows are mapped into Receive constructs
2. Output flows are mapped into Send constructs
3. Triggering and non-triggering flows determine the ordering of the receive-constructs. A triggering receive-construct is placed first.
4. Terminating and non-terminating flows determine the order of send-constructs. A terminating send-construct is placed last.
5. The content of flows are inherited into the PLD model, and can be accessed within the model.

Figure 9.12: Design knowledge for generating PLD models from PrM process.

Initial PLD models can be generated from the interfaces of bottom-level processes. The design knowledge for the translation is shown in Figure 9.12. The knowledge has been formalized in a set of translation rules. In Figure 9.13 we have shown some selected translation rules using the graphical notation from Section 5.4.5. Different patterns in a PrM model and their associated PLD constructions are indicated.

In Figure 9.14 we have illustrated how the process interface of P1.1 is translated into a framework of PLD model. We have omitted the detailed item specifications.

9.3.12 Extending the PLD Model

By translating a bottom-level PrM process to a PLD skeleton, the PLD editor is invoked. Here the PLD diagram may be extended to completely describe the process logic. A complete PLD model of P1.2 was shown in Figure 8.4.

Like other editors in the PPP tool, the PLD editor is syntax-directed. In addition to limit the user's ability to make syntactical errors, the editor provide facilities for creating, deleting, moving, resizing PLD constructs. Moreover, the editor support the user in invoking the process logic statements.

The developer is free to reorder the sequence of the receive and send constructs provided that the PLD structure corresponds to the logical port expressions. An
absolute condition is that the triggering flow is the first statement to be executed, and reversely, that the terminating flow appears as the very last statement. So, altering the structure of a PLD model may disturb its consistency with respect to the process interface. In the current version of the tool, updating of processes interfaces due to changes in the PLD model is not supported.

For the interested reader a further description of PLD editor is found in [153].

### 9.3.13 Ending the Modeling Session

By describing the logic of bottom-level processes using the PLD language, the PPP model is completely developed with respect to the modeling constructs offered by the PPP language. The product of the modeling session for our example is a PPP model with structure as shown in Figure 9.5.

This model will be basis for explaining the validation support provided by the
PPP tool in Chapter 10. Before we proceed, however, different features of the modeling environment of the PPP tool will be discussed.

### 9.4 Discussing the PPP Modeling Environment

In this section we will discuss the features of the modeling environment of the PPP tool. We will emphasize the modeling and verification support. Also the implementation platform, development strategy, and usability are addressed.

#### 9.4.1 Modeling Support

The modeling support is provided by the drawing editors of the PPP tool. It has been emphasized that the editor should be have uniform interface with a common “look and feel”. The basic features (e.g. "copy", "delete", "move", etc.) that resemble the manual “pen and paper techniques” are supported.

The PPP tool has been used to support a modeling exercise among our student mass. The purpose of the exercise was twofold. First, to evaluate the current version of the tool in a more realistic testing environment. Second, to get response for improvements and extension in future versions of the tool. The response was concentrated around area that are summarized below:

- During a modeling session unintentional errors like “typos” in text writing
may occur. To make a smooth correction of such errors an *UNDO facility* should be implemented. This enables the developer to cancel the last update — which was an error — and restore the previous model automatically.

- The modeling activities involves to some extent improvements of the layout i.e. straighten lines, align model objects, limit the number of crossing flows, etc. All these “polishing facilities” should be provided by the tool so that time-consuming routine work can be eliminated. Consequently, a flexible *gravity field* and algorithms for *layout of graphs* need to be implemented.

- Large models require large drawing space. In the current version of the tool the drawing space is unnecessary restricted. A larger drawing space is therefore required along with a flexible “zooming” function. In this way, both an overview of the complete PPP model and a detailed segment of the model can be provided.

An addition to these requirements, printing the model as *post-script* documents and automatic back-up of models were wanted.

### 9.4.2 Verification Support

The modeling session revealed the verification support provided by the PPP tool. This support serves two major purposes. First, it ensures the syntactic quality of the model. Second, it prepares the model for eventual translations. The following checks are provided:

**Implicit checks** are supported by the PPP tool. The PPP editors only provides the modeling constructs that are defined in the PPP language. Thus, morphological errors 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.

**Explicit checks** are supported by the tool in order to detect syntactical incompletenesses in the model. In Figure 9.8 the variety of explicit checks were shown. The user of the tool decides the scope of the checks as well as when they shall be activated.

**Semantic/consistency checks** are supported to some extent by the PPP tool. First, unique naming of PPP objects is checked. Second, in the PLD editor, PLD models are subject to type checking. For instance, the check ensures that items of a flow that enters a process has the same type as the ones that where sent in the sending process. Second, logical expressions that are formed by input and output ports are checked for inconsistencies. The check is performed both between processes on the same level of decomposition and between a mother process and her children.
9.4.3 Translations in the Modeling and Verification Support

Translations are used as an integral part of the modeling and verification support. As shown in the modeling session, two types of translations have been implemented in the PPP tool and exploited during the modeling and verification support:

Translations between Abstraction Levels in PrM The PrM model is usually constructed in a hierarchical top-down manner. Processes are decomposed into new PrM models when flows, ports, triggers and terminators have been attached to the processes. Their decompositions introduce a new level of detail, but these additional details are not to destroy the consistency between a higher-level process and its decomposition. Similarly, if a flat model is constructed (as is done in [151]), higher-level processes consistent with the flat model may later be introduced to form a model hierarchy.

Hence, we have introduced a translation in PPP that constructs a higher-level process from a PrM diagram, in which the flows, ports, triggers and terminators at this higher level are consistent with the interpretation of the lower-level diagram [8]. The translation is used to ensure consistency between different levels of abstractions.

Translations from PrM Models to PLD Models Bottom-level processes in the PrM model are specified by the PLD language. When the process logic of a bottom-level process is to be described, the environment translates the process to a skeleton of a PLD model that is consistent with the process interface. The PLD model may later be extended and modified as the system is designed, but a consistency check at the end of the modeling process makes sure that the final PLD diagram is still in accordance with its bottom-level process. Since it works on single bottom-level processes, it is possible to start modeling process logic in PLD while other parts of the system are still not specified in PrM.

9.4.4 Modeling Freedom

The work on PPP has emphasized finding a balance between the need for creativity in modeling and error-free models. Since the tool has an integrated philosophy and is directly supporting the PPP modeling approach, the developer is forced to follow this approach. Moreover, the drawing editors restrict the modeling activity to the languages provided by the PPP approach abandon the developer at making simple syntax error.

On the other hand, syntactical completeness checks are initiated by the user. As such, syntactical incompletenesses may exist as long as the user accepts them.
9.4.5 Development Strategy

The PPP tool is a result of work made by one or two permanent programmer’s supported by projects and diploma students. Since the major purpose of the environment was to experiment with generated ideas, the implementation didn’t follow a fixed, predefined specification. The requirements to the tool changed as new ideas were generated. In order to support the new requirements modifications and extensions of the tool’s functionality has followed very much a “code and fix” style of working. The major characteristics of such a development strategy can be summarized as follows:

- Several persons are developing the tool in parallel by realising different and often independent tasks.
- Each developer frequently needs a stripped version of the tool to realize and test out the ideas. Thus, several redundant versions of the tool exist in parallel.
- To integrate all approved ideas, problems might occur due to inconsistencies in the different versions.

These problem of managing the development are very much related to the ramifications provided by academic organizations. Thus, software development in such organizations is inherently hampered by its high turnover of junior personnel, unstable financial situation for research activities, etc.

Anyway, for eventual future development of the tool some guidelines can be given so that the above problems can be partly met. The development of the tool should be more rigorously managed and at the same time be flexible so that different versions of the tool can be tailored to specific needs. To meet these requirements a proposal for a development strategy is based on the principles from common system architecture [75] and incremental development [55]. A thorough discussion on how these principles can be applied on a future development of the PPP tool is given in [85]. The development guidelines are summarized as follows:

- Define a meta-model the PPP language. Both the domain-independent part and domain-specific language constructs should be defined.
- Implement the meta-model in a repository using a suitable storage medium — preferably a database.
- Decide the basic functionality of the tool — the core. Focus on common-used support facilities.
- The core should always contain a stable version of the current functionality of the tool and should be accessible for all the developers.
9.4. Discussing the PPP Modeling Environment

- During development, the various developers will need local versions of the core in order to test their tasks. Each developer will therefore either access parts of the core or the complete core and have his local add-ons which constitutes his tasks.

- When a developer has finished his task, the new functionality of the system should be a commonly accessed by expanding the core.

- Before the core is updated, there should exist a common consensus within the development team having discussed possible problems concerning the extension. Obviously, by extending the functionality of the core, there might occur integration problems with some parts of the old core. Consequently, some part of core have to be modified in order to meet the new functionality.

9.4.6 Implementation Platform

In the following we will discuss the implementation platform of the PPP tool. Particularly, we will evaluate its convenience focusing on the PPP tool as an experimental environment. Also, the platform will be evaluated by envisaging the tool as basis for commercialization. The main components of the platform will be discussed:

Hardware Configuration

The hardware configuration for the tool is very much decided by the configuration provided by our institute. When the development of first versions of the PPP tool started, only Sun work stations provided satisfactory graphical facilities. These stations have been the basic platform for the PPP tool since then. Our experiences with using the Sun work stations have been mainly positive. Some remarks should be given, though.

A crucial success factor for a tool its performance. Although the tool's performance depends on several factors, the hardware configuration plays a crucial role. In the main testing period of the PPP tool, Sun-3 stations with 4-8 MByte main memory were used. Using this configuration, the performance of the tool was satisfactory only for experimental purposes.

Minor experiments using Sun-4 with extended main memory (16 MByte) revealed a remarkably improved performance.
Windowing System

The PPP tool has been implemented under Sunview — mainly because of the existing configuration between PCE, BIM Prolog, and Sunview. Since Sunview is being phased out as a standard windowing system and replaced by X, the PPP tool is now only available in elder configurations of PCE, BIM-Prolog and Sunview.

Implementation Language

Using Prolog as the implementation main implementation has uncovered both its strengths and weaknesses.

Prolog allows the programmer to quickly test out new ideas. The declarative syntax of Prolog is convenient. In particular, the list mechanism combined with “recursive predicates” results in small and powerful programs. These facilities has been used extensively during the implementation of the PPP tool.

Furthermore, Prolog was exploited to define the storage format of PPP models. This simplified the storage facilities of the PPP tool. No additional mapping were needed in order to invoke a model representation into the modeling environment.

However, Prolog suffers from serious shortcomings for building large commercial applications. First, the work on the PPP tool revealed clear correlations between the implementation language and performance. As the size of the model increased and the tool’s functionality expanded, the performance diminished accordingly. A main reason for this is the non-deterministic search mechanism provided by Prolog. As the model and functionality increases, the search space increases, and it is more difficult to find the “correct” Prolog fact.

Another shortcoming of the PPP tool that is caused by Prolog is low maintainability. Although the development strategy discussed in the previous section, perhaps is the main reason for this obstacle, Prolog have limited mechanisms that encourage a structured and modularized code. This have resulted in a rather unstructured integration of code for user interface routines (PCE statements), functional logic, and storage routines. By exploiting modularity facilities in other structured implementation languages, the maintainability of the code could have been preserved.

A further discussion of the implementation platform for the PPP tool is found in [86].
Chapter 10

The PPP Tool — Translation, Execution, and Validation

In the previous chapter we presented how the PPP tool supports the construction and verification of PPP models. A result of the modeling session is a PPP model that covers the main parts of the bank domain. Now, we assume that the model is sufficiently specified so that it can be translated to an executable prototype.

This chapter addresses the three latter stages of the prototyping approach: translation, execution, and validation. Within the PPP environment several translation assistants has been developed. In addition to the two assistants described in the previous chapter, the following main translation assistants have developed in relationship with the PPP environment:

Translating PPP models to Simula/Demos By transforming PPP models to programs in Simula/Demos [62], behavior of real-world phenomena can be simulated. The PrM and PLD languages are slightly extended to accommodate this feature, especially for sampling of values from probability distributions. Using simulation models, non-functional properties like performance and reliability can be validated. Validation is supported by various traces and statistical reports generated during execution.

Translating PhM Models to SQL A model of the database is represented in PhM. Going directly from conceptual models to program code, the PPP environment transforms the PhM model to SQL statements that define the corresponding database. No additional statements are necessary to construct the database of the system. Since PhM resembles the traditional ER language, the transformation algorithm is almost similar to the ones used on ER models (see for instance [145]).

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 [74]. The translated code is complete and runnable, but in the current implementation, it is not integrated with the rest of the environment. The main motivation for including the transformation is to enable prototyping of user interfaces.

The translation assistants have been implemented by different diploma-students and project members. 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 10.1 and 10.2, respectively. The overall translation strategies are described and some selected translation rules will be included to the extent they shed light on the strategy. Details of the translation assistant for generating TEQUEL/C prototypes are explained using the compiler architecture described in Section 5.4.5.

# 10.1 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 [93].

## 10.1.1 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 PrM/PLD and Ada is exploited so that translations easily can be established. The most important arguments for translating PrM/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.
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.

Figure 10.1: The overall translation strategy for generating Ada code (from [93]).

- The PLD diagram is a block-structured algorithm that is easily transformed to a structured language like Ada.
- The ports in PrM (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 Figure 10.1. Details of the algorithm has been formalized as a set of translation rules. In Figure 10.2 we have shown some selected translation rules using the graphical notation for such rules from Broy [23]. 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 PrM/PLD model that are defined by the translation rules and translate them into their Ada counterpart.

It must be added that the translation schema is a bit more complicated than what the rules in Figure 10.2 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 Figure 10.2 should give a good indication of how PLD models are implemented as Ada tasks.

Figure 10.3 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
Figure 10.2: Some selected translation rules the "PPP-Ada assistant" (from [93]).

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 Figure 8.4 is also shown in the figure.

10.1.2 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\(^1\) sends a shut-down message that kills all tasks of the program.

\(^1\)Terminating task corresponds to a special process (PLD diagram) defined as 'terminating' in the model.
Figure 10.3: An overview of the Ada translation process.
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 Figure 10.4. Here we see that the process has an output port of AND type.

![Diagram showing a process flow and PLD model](image)

Figure 10.4: 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 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 Figure 8.4.

### 10.1.3 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.
10.2 Translating PPP Models to TEQUEL/C

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 Section 6.4 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 [56].

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 generated automatically, or semi-automatically with guidance from the system developer.

10.2 Translating PPP Models to TEQUEL/C

In this section we will take a closer look at the assistant for generating TEQUEL/C code from a PPP model. 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 [89, 88]. The interested readers is referred to [77] for a more complete documentation of the translation assistant. First, however, a brief introduction to TEQUEL.

10.2.1 TEQUEL

TEQUEL is a programming language with temporal semantics and has been developed at Imperial College, London [113]. The underlying execution platform
for TEQUEL is called the Rule Manager [138]. Roughly speaking, a TEQUEL specification is a set of rules on the form [78]:

\[
\text{formula about the future } \leq \text{ formula about the past}
\]

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 formula about the past is expressed in TQL, Temporal Query Language, whereas the formula about the future is expressed in TAL, Temporal Action Language [113].

In this work TEQUEL has served two main purposes. First, it has been used as a target platform for the translated PPP models. Secondly, it has been used to exploit the temporal semantics of the PPP language.

### 10.2.2 The Translation Strategy

The overall strategy for translating PPP models to C/TEQUEL programs is indicated in Figure 10.5. From a PPP model three separate parts are generated: (1) A TEQUEL part, (2) a C part, and (3) a Interface part which acts as an interface between the C part and the TEQUEL part. The translation strategies for generating the different parts are explained in the following. Some selected translation rules will be included.

![Diagram](image)

**Figure 10.5:** An overall architecture for the PPP-TEQUEL/C assistant (from [89]).
10.2. Translating PPP Models to TEQUEL/C

The TEQUEL Part

The TEQUEL Part contains the temporal statements based on the information given in the PrM model and the eventual database schema created from the PhM model. For prototyping purposes, this schema is represented as Prolog facts. The TEQUEL program consists of the following parts: (1) a declare part which converts information from the environment into a form that is manageable by the Rule Manager, (2) a never part that consists of constraints expressed in the conceptual model, (3) a query part that consists of derivation rules, and (4) a rule part which consists of action rules [113]. The current version of the code generator mainly exploits the declare and the rule parts. Generally, rules which can be interpreted from the PPP model are translated into TEQUEL rules that appear in the rule part. The effect of the TAL statement is specified in the declare part.

Translation rule a shows how a triggering flow and the triggered process is translated into a TEQUEL rule. The TQL statement says that when flow, containing item, appeared in the previous state, the TAL statement process(item) follows. The effect of the TAL statement is declared as an action that calls the C function cProcess (see below). Item can be sent from an agent, a timer or from another process, simulating both external events, temporal events and internal events, respectively.

Translation rule b shows the implementation of a delay. The rule that is produced says that if item is received a certain time ago (specified in the temporal expression Ticks), item is forwarded in flow unless the timer has been turned off in the meantime (receiving the off..flow). If flow is triggering a process, a rule like the one shown in translation rule a will also be produced.

The C Part

The C part corresponds to the internal behavior of the processes which are described by the PLD language. The procedural semantics of the PLD language makes a translation to C straightforward, giving one C function for each PLD model. Selection constructs, iterative constructs, and assignment constructs have the traditional semantics and are translated into the corresponding C statements.

As mentioned above, an action initiates a C function. A major part of the C part is devoted to transfer of data and control to and from the Rule Manager. All items that are contained in triggering input flows to processes are transferred to the function as parameters.

Furthermore, non-triggering flows between automatic processes correspond to data transfer between C functions. The actual transfer is implemented as a queue buffer. Translation rule c shows how a send construct produces an item that is put into the end of the buffer. The reception of item is shown in translation rule d. Moreover, the rule illustrates the handling of a triggering, repetitive flow. The
Translation rules

Figure 10.6: Some selected translation rules for the "PPP-TEQUEL/C assistant" (from [89]).

first item is passed to the TEQUEL part in order to trigger the process process. The item is received through the parameters of the C function. The remaining items is retrieved from the buffer structure.

Non-triggering flows from agents/manual processes to automatic processes and vice versa are interpreted as communication between the end-user and the system. Consequently, input and output routines are generated.

Flows from stores to processes are interpreted as queries, whereas flows from processes to stores work either as insert-, update-, or delete-statements depending on detailed specifications in the PLD model.
The Interface Part

The interface part takes care of the communication between the C part and the TEQUEL part by (1) statements which link created C functions to their counterparts in the TEQUEL part, (2) predicates which enable the sending of triggering flows to an event list used by the Rule Manager, and (3) predicates which enable the C program to access the temporal database. There are four kinds of accesses: queries, deletes, updates, and inserts. Translation rule e shows how a flow from a store to a process is interpreted as a single query and translated into a Prolog predicate. The item is defined by parts of the PhM model and is reflected in the predicate structure.

10.2.3 A Compiler View of the Translation

In the following, we will explain the implementation details of the translation assistant. To guide the explanation we will use the compiler architecture from Section 5.4.5. The error handler and the symbol table are discussed along with the appropriate phases of the architecture.

Lexical analysis

Lexical analysis aims at meeting the traceability requirement. This means that model names are as far as possible translated into equal names on the program level.

To assure that we have unique names in the PPP model, some of the names are slightly changed by removing spaces and line-shifts. In this way the produced code will correspond to the syntax of the naming conventions in the target languages. Alternatively, we could have enforced a correct model naming during the verification support. However, this would have constrained the modeling freedom unnecessary.

Syntax analysis

Syntax analysis performed at this stage addresses errors due to potential changes in the fact-structure during the lexical analysis. If two or more names are becoming equal due the lexical analysis, this will be detected during the syntax analysis. Furthermore, the syntax analysis has to take into account optional parameters that are invoked during the generation. It should be noted that before code is generated, the user must decide the length of a tick that defines the duration of the smallest time interval. This information is required for execution purposes and
utes” is a suitable tick unit. The tick size is an optional parameter that is checked during the syntax analysis.

Since the model information is represented as a Prolog fact, we are able to give an exhaustive list of errors without getting the problem of cascading errors. In cases of e.g. potential incompleteness and unreachability, the code-generation proceeds, but warnings are issued.

Semantic analysis

Here, type-checking is the major activity. This is performed on assignments, but no overloading is generated. Input/output routines uses the type information to perform type-dependent data input and output. Usage information on variables are kept track of to generate correct database calls.

Intermediate code generation

Most of the model representation is used directly in the translation process. However, some extra structures are generated, since they are more appropriate for code generation than for model representation used by the graphical tool. For instance, a specific control structure based on the triggering flows between processes are built to simplify the production of TEQUEL rules.

Code optimizer

So far, our goal has been to generate correct executable prototype code. Thus, optimization of the intermediate structures and the model information has not been emphasized. In order to generate efficient production-level code, this phase has to be emphasized.

Code generator

The actual generation of code is performed as pattern-matching. This is shown in Figure 10.7. Here, we depicted three translation rules concerning while-, assign-, and send-statement. The patterns in the PLD model and the corresponding C-statement are indicated. Furthermore, representation of the PLD model as Prolog facts is given. The translation rules are implemented as Prolog predicates. Based on the content of the PLD model, the suitable Prolog predicates are exploited. When the goal part of the predicates matches model fact, the content of Prolog fact will instantiate the remaining predicate variables in the goal part. After this
match, the action statements of the predicates are carried out and generate the defined C-statement.

Discussing the Translation

The largest difference between this way of producing code and traditional compilation is the specification representation pattern. Instead of having a strictly sequential ordering as in textual programming languages, the model information is represented as Prolog facts. This will ease the access to necessary information. Moreover, it means that we do not need to produce parse trees, as is done in traditional computers.

Furthermore, it should be noted that, just as with traditional compilers, the ordering of the phases is affected by options invoked by the developer. For instance, during the generation of the TEQUEL part, the developer is asked to specify the size of each execution step (tick). The specification of the tick size will govern the translation of timers.

After the translation is completed, the C and TEQUEL files are compiled in the traditional way. The object-files is then input to the Rule Manager that controls the execution. The execution sequence and how the executable program is used
Figure 10.8: Parts of the PPP model and the corresponding TEQUEL/C code. The execution sequence is labeled.

to validate the dynamic properties of the PPP model is described in the sequel.

10.2.4 Internal View of the Execution

The Rule Manager represents the prototyping environment and controls the execution. The execution sequence of the program is conceptually related to triggering flows of the PrM model. Figure 10.8 illustrates how process P1.2's PLD model and the P1.2's interface to other parts of the PrM model are translated into the prototype code using translation assistant described above. Here, the internal execution sequence of Rule Manager is shown on code level by labeling each execution step. The correspondence between the dynamic properties of the model is explained through these steps as follows:
10.2. Translating PPP Models to TEQUEL/C

1. Process P1.1 has been terminated correctly and internal event withdrawal\_transaction(X1,..,X5) is asserted in the Interface part.

2. During execution of the TEQUEL part, this internal event is registered and will trigger the rule with action verifyamount(X1,..,X5). This is recognized in the declare section of the TEQUEL part. On the conceptual level this step corresponds to the triggering of P1.2:Verify amount.

3. The action is a call to the C function verifyamount which corresponds to the PLD model shown in Figure 8.4. Now, the execution follows the logic of the C part. All items on the input flows are received.

4. Assuming that the C function evaluates the transaction to be valid. Then the C function sends information on the flow ok\_withdrawal. Information to the customer is shown by writing out plain text according to the definition in the send construct. As in step 1, this corresponds to an assertion of an internal event ok\_withdrawal(X1,..,X6) in the Interface part.

5. The execution of the process P1.4 will follow the same sequence as above.

This internal execution sequence is hidden from the user of the system. The user view of the execution is explained in the next section.

10.2.5 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.

Assuming that the initial database content is as indicated in Figure 10.9a. 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 (minutes) which should cover the period of processing a transaction. During the period the user invokes the
external events that are shown in Figure 10.9b. 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 (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 describes 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 Figure 10.10, 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 Figure 10.11, 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 Figure 10.9a 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.
The Final Temporal Database

1. customer(Odd Ivar,1001,0,99999)
2. account(1001,5612,0,99999)
3. happened(transaction(1,1002,6000,W),1,1)
4. verifytransaction(1,1002,6000,W,2,2)
5. invalid_transaction(1,"error",2,2)
6. happened(transaction(1,1001,6000,W),3,3)
7. verifytransaction(1,1001,6000,W,4,4)
8. withdrawal_transaction(1,1001,6000,5612,4,4)
9. verifyamount(1,1001,6000,W,5,5)
10. withdrawal_rejection(1,"error",5,5)
11. happened(new_withdrawal(4000),6,6)
12. ok_withdrawal(1001,4000,5612,7,7)
13. withdrawamount(1001,6000,7,7)
14. withdraw_result(1001,8,8)
15. new_balance(1001,8,8)
16. account(1001,1612,8,99999)
17. issue_account_st...(1001,1612,9,9)
18. process_transaction(1001,1612,9,9)
19. account_statement(1001,1612,9,9).

Figure 10.11: Parts of the temporal database after execution.

10.2.6 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 on the 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, the internal logic of automated processes is specified by PLD models. Detailed specifications may hamper the validation process if rapidness is a central requirement. In TEMPORA, process logic may be specified as business rules using the External Rule Language (ERL) [138]. Moreover, this language is used for specifying constraints on the data model. A translator from ERL to TEQUEL is currently being developed at Imperial College. ERL will therefore be an interesting alternative to the PLD language, since the business rules are expected to be captured regardless of design considerations. If the process logic is completely specified by ERL, the generated code will consists of only the TEQUEL part. Such an approach is expected to be more in accordance with rapid prototyping than the current approach.

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 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 [143], 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. 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 PrM construct. Moreover, external events can be entered in a “pop-up” window and temporal results of the execution could be accessed by “clicking” one the relevant PrM 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 [143] 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. 10.10, 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 granular-
ity 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.

Improvements of the prototyping environment is dependent on future versions of the Rule Manager. This discussion, however, is outside the scope of this text.

10.3 Comparing the Translation Assistants

This chapter has presented two works for translating PPP models to executable prototypes. Most emphasis has been on discussing the execution semantics in the PPP model that is reflected in the execution semantics provided by the target language. The correspondence is realized in the translation rules that are shown in connection with each work. Implementation details of the translation strategy has been explained in connection with generation of the TEQUEL/C code using the traditional compiler architecture.

The different works have revealed different validation potential. The prototyping environment provided by Ada restricts the validation to program testing. The Rule Manager used to execute TEQUEL/C prototypes have more sophisticated mechanisms in order to encourage user involvement. A tighter coupling between the PPP tool and the Rule Manager would have enabled animation, though.
Chapter 11

Conclusion

This chapter concludes the thesis. The major results and achievements in the thesis are summarized in Section 11.1. Directions for future work are outlined in Section 11.2.

11.1 Major Achievements

In the following the most important achievements of this thesis are listed. They point out what has and what has not been done by the author. Also, the original contributions of the thesis is discussed:

- The context of the thesis is established by the author. It relate methodological components and tool components that are known from the literature. Their relationships and mutual effects are discussed throughout the thesis.

- The basic motivation and terminology described in Chapter 2 are known from the literature. It should be noticed that the way conceptual modeling is explained, conceptual models relate the realms of IS engineering and knowledge engineering. The results from the thesis can therefore be applied in both realms.

- There are existing proposals of taxonomies for discussing quality of conceptual models. The idea of building a taxonomy which separates quality goals and quality means at the model level is the author's. Sindre has contributed to the terminology used in the taxonomy [92].

- The distinction between different approaches for improving model comprehensibility is a result of the a collaboration between Gulla, Seltveit, Willumsen, and the author. A paper discussing the various approaches and how they can be combined has already been published [155].
• The idea of regarding translation of conceptual models to traditional programming languages using a compiler architecture explicitly is the author's. The translation assistant from PPP models to TEQUEL/C made by Krogstie is evaluated using this architecture. As combination of the two results is published in [88].

• The PPP language has been developed by several researchers and students within the Information Systems Group. The author has contributed and is a co-author of three papers/reports [58, 90, 91].

• The first versions of the PPP tool were implemented by the author [84]. Thus, the architecture and functionality was initiated by the author. A newer version of the tool has been realized in collaboration with several project members. In particular, Mingwei Yang has contributed to the current version of the PPP tool [157].

• Willumsen and Gulla have contributed to the PLD editor and translation to Ada code, respectively. This translation assistant has been evaluated from a validation point of view and published in [93].

• The translation of PPP models to TEQUEL/C code was implemented by Krogstie. This assistant is discussed from a validation point of view in [89], and published with Krogstie as a co-author.

11.2 Direction for Future Work

The thesis has documented that it is realistic to apply a prototyping approach to validation within CASE environments. Several areas for further work can be envisaged. The directions discussed in the sequel are based on the various observations and evaluations that have been carried out in connection with this thesis.

11.2.1 Evaluating the Taxonomy for Model Quality

Our framework for model quality is based on the assumption that the properties of conceptual modeling languages (and implicitly their support tool) affect the potential for assessing the quality of the associated models. The proposed relationships are argued for and exemplified throughout the thesis. Even so, further evaluation of the taxonomy is needed. Controlled experiments and comparative analysis should more thoroughly discuss the taxonomy and devise possible changes.

Today, the framework is purely qualitative. Quantitative measurements of model quality can be introduced to estimate the costs/benefits of bringing a model up to a certain quality level. Extensive experiments are needed to establish quantitative measurements.
11.2.2 Towards Evolutionary Prototyping

The translation assistants presented in Chapter 10 are limited to production of correct and executable code. This means that the generated prototypes ignore software performance concerns. The use of the prototype is therefore restricted to model validation and the software prototype is meant to be “thrown away” after the model is validated. The model is nevertheless preserved.

Having a long term objective of supporting the complete life-cycle requires that we can generate efficient production-quality code. Then, the optimization aspects of the translation assistants have to be addressed. Detailed design knowledge must be represented in the translation rules, so that modular and efficient code can be generated automatically or semi-automatically. A proposal for generating call-structures from PrM models [156] is a natural starting point. Also, efficiency factors for generating Ada code should be considered [56].

So, by improving the quality of the generated code with respect to performance, maintainability, etc. the prototype should have the potential of evolving into the final system.

11.2.3 Towards an Integrated Prototyping Environment

The two translation assistants described in Chapter 10 generate prototypes that are executed by environments that are not integrated with the PPP tool. Only a subset of the various execution mechanisms described in Section 6.4 is provided by these environments. By tailoring the prototyping environment to the rest of the modeling environment, a wider range of mechanisms can be provided. A step in this direction is the integrated environment for interpreting conceptual models that is proposed in [154].

11.2.4 Towards an Integrated Validation Environment

Even if the prototyping approach is emphasized in this thesis, two alternative approaches for improving model comprehensibility were outlined in Chapter 5. Ongoing doctoral thesis work addresses these validation techniques. The complexity reduction approach is documented in [125], whereas the generation of execution traces combined with query and explanation facilities is presented in [60].

Further work would emphasize integrating these techniques. How three different techniques can be integrated is described in [155]. The architecture of this integration is illustrated in Figure 11.1. The core of the architecture is the PPP conceptual model. This model can be translated into an executable prototype, a model view, and/or an explanation. By executing the program an execution trace is gener-
ated. The trace contains information about the dynamic properties of the model. Information from this trace is used to produce explanations. This explains the observed behavior in response to questions posed by users or developers. This explanation module may also generate different views of the model. Combining the pictorial view of the model and the textual explanation, multimedia explanations are supported.

![Diagram](image)

Figure 11.1: Integrating three validation techniques (from [155]).

By integrating different validation techniques within a coherent framework, comparative experiments could reveal their relative strengths and weaknesses. Joint effects of combining two or more techniques may be determined.

11.2.5 Integrating Configuration Management

The PPP modeling approach is incremental and iterative. The PPP models are developed through a combination of modeling, verification, translation, validation, modification, etc. Although most parts of the approach is supported by the PPP tool, some model errors (invalidities) may not be detected prior to translations. These errors are propagated during the translations and make the modifications cumbersome without effective tool support. Furthermore, a depth-first approach to modeling leads to situations where different parts of the model/system are developed to different levels of detail at the same time. Finally, future development environments must envisage a multi-developer situation where different parts of the model are developed by developers that are located at geographically different places.
11.2. Direction for Future Work

All these factors require an integration of configuration management in the PPP environment. The elaborations and specifications concerning model versioning, version numbering, group development support, project management issues, efficient version storage, etc. given in [4] should be realized within the PPP tool.

11.2.6 Comparative Experimental Surveys

The potential of the PPP approach has been evaluated based on limited case experiments. To really document its potential, it has to be tested in more realistic industrial settings. However, a major prerequisite is that a more polished prototype version of the PPP environment is built. The version should meet the desiderata which are given to early releases of commercial products concerning performance, operational stability, user-friendliness, etc.
Bibliography


