COMIS – A Conceptual Model for Information Systems

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

This thesis reports the work of an improved modeling approach for information systems. The motivation of the work comes from the PPP project which aims at developing a prototype of a CASE tool for information system engineering. During this project, it became clear that modeling techniques, especially the formal concepts and verification methods, play a key role in establishing a solid foundation for any CASE tool.

The work includes a study of the state of the art of contemporary modeling methods, which results in a proposal to define a set of criteria for desired features of modeling methods; an integrated modeling framework (COMIS) that consists of a data model (ONE-R), a process model (PPM), an actor model (AM) and the cross references among the three sub-models of the COMIS modeling framework; and a case study that specifies the IFIP conference example by the conceptual components of COMIS.

The kernel of this thesis is COMIS, which contains a set of formal components that enable more accurate specifications of information systems. COMIS also removes the restriction of those modeling methods that often regard an information system as a response system, so that information management in a wider scope, say, organizational business systems, may be modeled. The thesis also includes several consistency check methods that may be used to verify the correctness of a system specification.
Acknowledgement

The work of this thesis has been being done with great help of many persons to whom I express my gratitude:

- First, I thank professor Arne Sølvberg, my supervisor, for his continuous guidance and encouragement. He helped me to made the study plan, discussed problems with me regularly, provided many valueable suggestions and criticizms, and kept on reading the drafts of the thesis with patience and enthusiasm. Besides, it must be pointed out that his early work on semantic data modeling is an important source for the studies reported in this thesis.

- I am also thankful to my fellow doctorate students in the Information Systems Group: Rudolf Anderson, Gunnar Brataas, Jon Atle Gulla, Odd Ivar Lindland, Anne Helga Seltveit, Vidar Vitland and Geir Willumsen. We work in the same field, so the discussions among us have been extremely helpful. In particular, Lindland has been cooperating with me in the PPP project which has resulted in the prototype of a CASE tool.

- I also want to express my gratitude to two former members of the group: Dr. Kung Chen He and Dr. Guttorm Sindre. The consistency check methods presented in my thesis are developed on the basis of their early work, so I owe them a lot.

Trondheim, August 1993,

Mingwei Yang

Mingwei Yang
Preface

The problem statement

In contemporary books on information system development, the word “modeling” often has very high frequency of use. Indeed, modeling techniques have become extremely important in any new system development methodology – students must learn them, engineers must use them, and researchers must study them.

The importance comes from man’s fundamental way of thinking when observing the world: abstraction. In order to understand, analyze and create complex phenomena or systems, the best way is to concentrate on common properties and aspects while ignoring subtle differences. The method of abstraction is called modeling. The result of applying a modeling method to system’s problem is a system representation called a model. When the model of a system focuses on those aspects that are particularly important to us, it brings us better understanding of the relevant phenomena of the systems.

The principle of abstraction is also applied in software and information engineering, especially when the systems are large and complex. Therefore, when the word “software crisis” appeared in 1970s, modeling techniques, including early data models [36, 26, 37, 146] and data flow diagrams [39, 59, 157] (DFDs) were developed to support software engineering. In the 1980s and the early 1990s, more efforts have been made for improving the modeling methods. Some of the results can be found in the series of proceedings of IFIP conferences [142, 143, 41, 53, 137, 42] and in other literatures. “The concept of modeling is inherent in any information systems methodology” [70].

Modeling methods may be categorized according to different classification schemes. Some methods have already been commercially marketed, e.g., E-R, DFDs, JSD, etc.; others are still being researched.

- The industrialized modeling methods have provided practical and effective means in analyzing and building information systems. These methods are also used for training engineers and researchers in information system development. The students learn to understand the key features of a system by applying appropriate method knowledge.

- The modeling methods that are still in the realm of research provide
elaborately chosen modeling concepts in order to enhance both the expressiveness and accuracy of their associated specification languages. Many modeling approaches have strong mathematical foundations, so a system model may be represented by a set of mathematical formulae. Such system specifications are formal, so their properties and correctness (consistency) can be proven by mathematical means, e.g., inference. Since the correctness of a system model is a crucial factor for the success of the development, consistency analysis has a high priority in research on information systems modeling.

Can we enjoy the benefits from the two sides at the same time? Unfortunately, at present the answer is no. The reality of modeling techniques is:

- The traditional modeling methods used by the currently accepted approaches are often informal, so the meaning of the conceptual components of a system model may be vague and ambiguous. The lack of formal specifications makes detection of errors difficult. This is unfortunate since any error in the early stages of developments may bring about much trouble later on.

- The newer modeling approaches often concentrate only on limited aspects of information systems. They may also fail to provide efficient means for consistency checks, even if many of them are formal approaches. These weak points prevent their wide-spread applications to the engineering practice.

There is still a gap between theoretical researches on system modeling and the requirements of information system engineering. CAN WE FILL THE GAP? In other words, can we develop a modeling method which is powerful both with respect to formal description capability and with respect to the applicability to engineering practice? To answer this question is the major motivation for the work reported in this thesis.

Another challenge comes from the significant changes in the technical basis of information systems. In the 1980s the object oriented paradigm has influenced and dominated the fields of computer software studies. Meanwhile software manufacture has become an industry. A software market has developed dealing with software as a “commodity”. One consequence is that the concept of “reuse” has become important. The “old” development methodologies had the systems built through detailed programming, analogous to building a house from bricks. Now we have bigger “blocks”, so the main tasks have changed to “assembling” the house from the pre-fabricated components! CAN A NEW MODELING APPROACH REFLECT THIS CHANGE? To answer this question is also a major motivation for us to try to develop a new modeling approach.

In this thesis an integrated modeling framework is proposed. The conceptual components in the framework are defined. A set of consistency check methods is also developed. The specification model has been partly implemented in the
experimental CASE tool PPP which has been developed by the author and other doctoral students in the Information Systems Group.

The central topic of the thesis is improving modeling techniques for information system engineering. Two main issues are discussed:

- An integrated system modeling method with its associated components and/or specification languages;
- The formal methods to verify or check consistency of system models.

**Approach**

The term “information system” has been used for a wide range of systems, so it has a very general meaning. No modeling method has the capability of fitting all types of system problems. Our work starts with an analysis of the types and scopes of systems, and makes a classification of them. On this basis, we determine the categories of systems for which the new modeling approach is applicable:

- the “database centred” systems without strong time dimension;
- the systems cooperating within whole or some parts of an organization to support its business functions, i.e., multi-application systems.

More explicitly, the modeling approach is parallel to the conceptual framework given in the interim report by the IFIP WG 8.1 task group FRISCO [45]. In order to specify the information processing activities within an organizational system which behaves as an open active system (OAS), our modeling approach will cover several modeling perspectives:

- data modeling which specifies and analyzes the properties and structures of data, i.e., the operands of the information processing activities within the OAS;
- process/procedure/rules modeling which specifies and analyzes the information processing activities;
- actor modeling which specifies and analyzes the roles of the organizational units, persons and software packages which participate in the information processing activities.

The relationships among the conceptual components in the different perspectives will also be defined so that a system model is an integrated system model.

Another part of the thesis work is to develop consistency check methods that not only take advantage of formal system specifications, but also take into consideration the feasibility of the associated proof procedures. Different methods are associated with the different parts of a system model:
• meta modeling – using the data model itself as the meta model to check the structural consistency of the whole system model;

• mathematical logic – using the formal ways of the first order logical system to verify the consistency of the specifications that are produced by modeling;

• state-transition-diagrams to check the constructivity of process hierarchies.

Besides the formal methods for consistency verification, a case study is used in order to validate the improvements that we claim to have made with the new modeling approach. Part of the model has been implemented in an experimental CASE tool called PPP which has been developed in the Information Systems Group. A brief introduction to the PPP tool is included in the thesis.

Major results achieved

The new modeling approach is based on the work done by the members of the Information Systems Group over several years. The work reported in this thesis makes contributions as follows:

• redefine the original data model (the PM model) to

  – support complex data structure;
  – combine the features of the ER approach and object oriented data models;
  – provide both the user friendly conceptual constructs and formal internal type expressions;
  – support the reshaped relational algebra for the complex data structure and the semantic data language SDL which are defined so that the operations can be described in terms of the conceptual model directly;
  – support the concept of scenario which supports a three-level schema architecture to multi-application views;

• redefine the original process model (the PPM model) by

  – adding formal constructs to the data flow diagrams;
  – combining it with the actor modeling so that reusable software components can be described and merged with the process model, and so that processes can be grouped into actors (composing application systems) applying extended object oriented methods;
  – defining a canonical port structure as the foundation for checking the consistency of the process hierarchies;
• develop a set of consistency methods that check a system specification as a whole or check a part of it;

• develop many parts of the PPP tool.

The outline of the thesis

The thesis is divided into three parts. The first part makes the preparations for describing the new modeling approach: in Chapter 1 the basic concepts are described aiming at specifying the problem domains in which the discussions of the thesis are carried on, and providing a unified set of terminology. Chapter 2 is an investigation of existing modeling methods. In the same chapter we also analyze these models and summarize their strong and weak points according to the criteria we defined in Chapter 1. In addition to illustrate the state-of-the-art of modeling techniques, we also reveal the problems that we intend to solve with the new modeling approach.

The second part reports on the efforts to improve the modeling techniques for a category of information systems. In Chapter 3 an integrated system model called COMIS (CONceptual Model for Information Systems), including a data model, a process model and an actor model, is presented. Exploiting the formal properties of the proposed model, Chapter 4 gives the methods to check the consistency of a conceptual schema constructed by the modeling primitives. In Chapter 5, the PPP project and the associated modeling environment are introduced.

The last part summarizes the new modeling approach both by validating it through a case study and by analyzing it according to the criteria proposed in Chapters 1. The work is presented in Chapters 6 and 7. In Chapter 7, we conclude by giving a general evaluation of the modeling approach as well as providing some suggestions for further work.
Contents

I  Preliminaries on information system modeling  1

1  Basic concepts  5
   1.1  Terminology  5
   1.2  The concept of information system  7
   1.3  Information system modeling  14
      1.3.1  The role of conceptual modeling  14
      1.3.2  The ontological approaches  15
      1.3.3  What is desired for a better conceptual modeling method?  21

2  A survey of modeling methods  23
   2.1  The Structured Analysis approach  23
      2.1.1  Basic components  24
      2.1.2  Leveled data flow diagrams  24
      2.1.3  Evaluation  25
   2.2  The Entity-Relationship Model  26
      2.2.1  Basic components  26
      2.2.2  Evaluation  27
   2.3  More semantic data models  28
      2.3.1  Basic components and concepts  28
      2.3.2  Evaluation  30
   2.4  The Jackson System Development  31
      2.4.1  Basic components and concepts  31
      2.4.2  System modeling  32
      2.4.3  Evaluation  33
   2.5  SSADM  34
      2.5.1  Basic components and concepts  34
      2.5.2  System modeling  34
      2.5.3  Evaluation  34
   2.6  IE  36
      2.6.1  Basic components and concepts  36
      2.6.2  Systems modeling  37
      2.6.3  Evaluation  38
   2.7  BNM and REMORA  39
      2.7.1  Basic components  39
      2.7.2  The conceptual modeling of systems  40
      2.7.3  Evaluation  40
   2.8  Object-oriented modeling  42
      2.8.1  Basic concepts and system modeling  43
II The new modeling approach

3 COMIS – an integrated system modeling approach
3.1 The modeling framework 51
3.2 The Extended Entity-Relationship Model (ONE-R) 56
3.2.1 Introduction 56
3.2.2 Basic components 58
3.2.3 The Formal Expressions of ONE-R 63
3.2.4 The Algebra of ONE-R 73
3.2.5 The Semantic Data Language 80
3.2.6 What is new in the ONE-R model? 87
3.3 The Process Port Model 88
3.3.1 Introduction 88
3.3.2 Basic concepts 90
3.3.3 Substitution of or ports 95
3.3.4 Substitution of xor ports 97
3.3.5 Reusable definitions 104
3.3.6 More auxiliary concepts for other components 106
3.3.7 The cross references between PPM and ONE-R 107
3.4 The Actor Model 109
3.4.1 Introduction 109
3.4.2 The modeling concepts 111
3.4.3 The mapping between AM and PPM 112
3.4.4 The cross references between AM and ONE-R 114
3.5 Brief Summary 115

4 Consistency verification 117
4.1 Transforming an ONE-R DB schema into a logical theory 119
4.2 The meta model 124
4.3 Formal verification for ONE-R 127
4.3.1 Lewis’s work 128
4.3.2 Kung’s work 129
4.3.3 The consistency verification for ONE-R 132
4.4 The static consistency check for PPM and AM 133
4.5 The dynamic consistency check for PPM 137
4.5.1 Some possible ways for checking the constructivity 137
4.5.2 Construction of an STD through Canonical Process Ports 141
4.5.3 Synthesis for the Properties of the Process Network 154
4.6 Brief summary 157

5 PPP – a CASE tool to support the modeling 159
5.1 The working environment of PPP 159
5.2 The structure and repository of PPP 160
5.3 The modeling facilities in PPP 161
III  The evaluation of the new approach  167

6  A case study  171
   6.1  Problem description  ......................... 172
   6.2  The data model  ............................. 174
       6.2.1  The basic scenarios  ................. 174
       6.2.2  The external scenarios  ............... 178
   6.3  The process model  .......................... 180
       6.3.1  The top level processes  ............. 180
       6.3.2  The decomposition of the processes  190
   6.4  The actor model  ........................... 200
       6.4.1  The actors and their functions  ....... 200
       6.4.2  Mapping the actors to reusable processes 204
   6.5  The cross references  ....................... 207
   6.6  Brief summary  ............................. 207

7  Concluding remarks  209
   7.1  Comments on the modeling approach  ....... 209
   7.2  Further work  ................................ 211

A  The SDL Syntax  213

B  The proofs for theorem 3.1, 3.2 and 3.3  221
Part I

Preliminaries on information system modeling
In Part I we give the preliminaries on modeling techniques for information systems. We will specify the problem domain of our work, the state-of-the-art of the field, and the problems we intend to solve.

The contents of Part I are organized into two chapters. In Chapter 1 we discuss some basic concepts of information systems. We develop a set of definitions that is used throughout the thesis. Based on these definitions and associated discussions, we propose a set of criteria to determine desirable properties of a "good" modeling approach. These criteria will be used to evaluate some existing modeling approaches as well as to evaluate the approach reported in this thesis.

In Chapter 2 we investigate several contemporary modeling techniques in order to show the state-of-the-art of modeling techniques. The methods include both some that have been commercialized and some that are still in the research realm. We comment on them, and point out the problems that we attempt to solve by an improved modeling approach.
Chapter 1

Basic concepts

In this chapter we describe and discuss a set of concepts that are relevant to information systems. Definitions of the concepts can be found in the literature [28, 20, 70, 66, 80, 89]. There have even been two IFIP WG 8.1 conferences particularly held to discuss information system concepts for improving the understanding of information systems [41, 42]. However, there is no conclusion emerging in the literature. The definitions of many concepts are never given in a unified form, although several definitions for one concept may have the same meaning. Hence, we have to define some concepts in the terminology system which is to be used throughout this thesis in order to build a foundation for constructing the modeling framework.

In particular, we will discuss the concept “information system”. This concept has been explained differently by different people. Our discussion will result in a conceptual framework which will be used to evaluate some existing models and will also serve as the starting point to our proposal of an improved modeling approach.

1.1 Terminology

In a fast developing scientific and technological field, the problem of terminology is often troublesome. This is certainly too in the field of computing. In order to keep the terms we use in this thesis in a universal way, a terminology system must firstly be built.

We therefore give the following definitions:

Information system: A system which provides management for collecting, storing, processing and distributing information within an organization.
This general definition does not mention if the work of an IS is done by computer or man, so the IS may consist of automatic parts as well as manual parts. If an IS is supported by computer hardware and software, we call it a computerized information system (CIS). Our major interests are on CISs because building computerized information systems is the main objective of information system engineering.

**Software system:** A system consisting of computer programs that can be executed by a computer.

The reason that we define this concept and distinguish it from the concept of information system is that they are easily confused. We are often told that some company is selling an information system or has developed an IS, but in fact the company is only selling or developing a software system that can be used to build an information system. A concrete example is the difference between a DBMS and a database that is built with the help of a DBMS.

It is worth emphasizing the difference between the two concepts in order to understand the different objectives of software engineering and information system engineering. This has been illustrated in [99]. Here we want to emphasize that an information system which manages information within an organization can never be bought from the software market, but can be built by using some hardware & software systems available in the market.

**System development methodology:** An explicit way of specifying systems and organizing the activities for developing systems.

Many methodologies for information system developments have been proposed. Most of them include guidelines on how to divide the process of building an IS into clear phases and how to specify a system during each phase.

Most methodologies have divided the process of development into phases like planning, system analysis, system design, system implementation and system maintenance. Each phase may be further divided into smaller steps. As an example, SSADM [29] has such a development process. Since this way of system development follows rigid time sequences, it is characterized as a "water fall" development life-cycle. There have been some criticisms to the "water fall" paradigm, so some different development process models, e.g., the "spiral model" [72, 73], have been proposed to overcome the weakness of the water fall model.

The second aspect is to describe the system at any specific time point in the development process. We often call such a description a model of the system. This concept is defined below.

**Model:** A specific representation of an object at some level of abstraction. A model always reflects some special properties of the object and ignores the details that are deemed to be unnecessary.
1.2. The concept of information system

Modeling: The "creation" (or "building") of models.

In our research field, the term "model" is often given different meanings for different purposes:

1. A set of **meta concepts** which gives the conceptual framework, i.e., the conceptual primitives and the constructors used to describe a system. For instance, the relational data model contains the concepts of relations, attributes, etc., that are used to specify the structures and features of relational databases.

2. The representation of a system. The model is built by using the meta concepts to specify a concrete system. Such a model is also often called a **schema** of the system, or the **specification** of the system.

3. The state of a system. When we regard the schema of a system as a logical theory, then a legal state of the system is called a model of the schema. A schema which has at least one model is called a **consistent** schema.

Apparently, conceptual models have a taxonomy structure which has been discussed in [55] where four levels of concepts are illustrated. In this thesis, we mostly use the term "model" with the first meaning (meta-model), but also use it with other meanings when the context is clear.

1.2 The concept of information system

Although we have given a definition of the concept "information system" in the previous section, we will discuss it further. Since the definition has a quite general flavor, people's understanding and explanation of the concept of information system may be very different due to different interests, technological backgrounds, application domains, etc. In an IFIP WG 8.1 report which tries to change the situation that "many fuzzy or ill-defined concepts are used in the information system area", it is stated that

> "it became obvious that the concept information system itself is not a single and unique one. Information system is interpreted quite differently by different groups of people. A number of such interpretations have been identified, but so far they are not fully understood." [45]

No doubt the differences in the understanding of this basic concept will impact on the modeling approaches for information systems. It is such differences that may mead us to the sources of some of the problems of many modeling
approaches, and thereby motivate the work for the modeling approach reported in this thesis.

In the early 1960s, the term of "information system" was used to denote technical devices, mainly consisting of one or more computer programs, that helped people to capture, store and manipulate data for various purposes. This perception was held in a quite long time. As an evidence, in Connor's book (written in 1985) that describe and compare some system development methods [28], the following definition is given:

"An information system is a system that provides management with human-readable information."

In this book, the basic components of such a system are illustrated as those shown in Figure 1.1.

![Figure 1.1: The system as a “Black Box”](image)

Such a view reflects the early consensus of people about what an information system should be. This view has deeply influenced many efforts to make and improve information system development methodologies over many years.

In order to support our statement, we list some commonly used features of information systems:

- **System boundary.** It is considered very important to determine what is a part of a system and what is not. This is accomplished by defining a boundary of the system, i.e., anything inside the boundary is regarded as part of the system and anything outside the boundary belongs to the environment of the system.

- **System functions.** A system has a set of functions that is needed to meet the requirements of some clients in the environment of the system. Internally, the functions can be further decomposed into more sub-functions; however, all the sub-functions are organized to support the functions of the system as a whole toward its environment.

- **System behavior.** A system will carry some actions when some events happen. Very often, the actions are triggered by some external events,
also called *stimuli*, from the environment, and the actions are called *responses* since they also produce effects on the environment upon receiving the stimuli. By the way, we often call such systems *reactive systems* or *response systems*.

- **System inputs and outputs.** When an action is triggered, the system usually get some information, called *inputs*, from some external entities in the environment, and/or send out some information, called *outputs*, into the environment. The possible inputs and outputs of a system often compose important parts of the system functions.

- **System states.** A collection of definable properties of a system is called the *state* of the system. The state of a system may change from time to time, often because of the effect of external events, and the associated system actions.

The properties described above are typical of technical systems, particularly of software systems that consist of one or several programs and work on a set of data, because

- a program has a very clear boundary, often called *user-interface* through which the system interacts with its users;

- a program always provides some functions to its users directly or indirectly;

- a program usually is in a static state when nobody uses it, but reacts to user's messages whenever it is invoked. Most programs are reactive systems;

- a program must get inputs and produce outputs in order to perform its functions;

- the data set managed by a program is always in a specific state; the state changes with the execution of the program.

Such a system can also be illustrated as shown in Figure 1.2.

However, such a view had exposed its limitation even as early as in the 1970s. When the data base techniques were developed, the concepts of *data sharing* and *data independence* [37] and the corresponding facilities provided by DBMSs made it possible to let several systems for different applications work on a same database(Figure 1.3). This kind of configuration has exposed some problems of the basic components and key features of information systems:

- **System states.** Obviously the state of the database can no longer be viewed as the state of an single application system, because it only manages part of the database. Even the part of database that a particular
Figure 1.2: A "reaction" IS

Figure 1.3: Application systems on a DB
1.2. The concept of information system

application system is able to access can not be viewed as the state of that application system, since other systems may be able to access the same part of the database;

- **System boundary.** It becomes more difficult to define the boundary of each system. Is application system 2 outside the boundary of application system 1? The answer is hard to give since the two systems influence each other’s behavior, but not in the manner of stimulus-response. They just interact implicitly through the shared database;

- **System functions.** Should the database and all the application systems working on it be regarded as one system? It seems to be no problem because the systems are already linked together through the database. However, it becomes difficult to define the functions of the “big” system, not only because the functions of each application system have been defined for specifical purpose, but also because the systems are not necessarily built at the same time. Instead, they are often built one by one, and the oldest system may not be aware of the existence of the newer systems.

Furthermore, even the type of system configuration shown in Figure 1.3 was soon proven to be out of date. During the 1980s and the early 1990s, the progress of computer technologies has been so tremendous that the way people manage their information may be drastically changed.

Two aspects of the progress are particularly worth noting: 1) the rapid development and wide application of computer networks that make it possible to link people and systems across organization units, regions and even different countries; 2) the fantastic development of micro-processors that resulted in millions of high-performance, low-priced personal computers and work stations pouring into society.

One result of this is that more and more people, not only computer specialist, but also executives, managers, and other staffs, are using computers for a wide range of information management tasks on distributed or decentralized technical platforms; more and more systems, previously isolated, are linked together and cooperating through networks; more and more built-in software packages with standard interfaces are available in the market and can be merged into a network of systems at any time. Accordingly, more terms about the information system, such as “end-user information systems” and “Intelligent and cooperative information systems” (ICISs) [16] have been used.

Such a system configuration, as that shown in Figure 1.4, becomes far more complex than that shown in Figure 1.1. The systems, while each itself is regarded as an information system with its own specific functions, form a larger composite system as when they are connected and/or cooperating. The features of the system are certainly quite different from those that we have
listed above. In addition, humans, as “intelligent agents” [16], are now playing more important roles in such a system.

Figure 1.4: Technical systems in an organization

All these changes have forced us to re-consider the concept “information system”. For example, some definitions, such as “An information system is a means of recording and communicating information to satisfy the requirements of all users, the BUSINESS ACTIVITIES they are engaged in and the OBJECTIVES established for them” [70]; “A system is an association of independent devices, people, rules and/or procedures organized to form an integral whole to achieve a common purpose” [28, 81]; etc., reflect the changes in people’s view on information systems.

In Essink’s papers [47, 48], the views of IS are classified as the data transformation view, the information flow oriented view, the information base view, the problem solving view, and the embedded social system view. They are recognized by people in several phases or periods. However, in my opinion, the shift to a “newer” view does not mean that “older” views are no longer useful. Since modern information systems are becoming more complex with compound and hierarchical structures, they should be observed and analyzed on different aspects, at different levels, and according to different views. This has been pointed out in the FRISCO report [45]:

“The term ‘information system’ seems to be used in at least four different senses:
1.2. The concept of information system

- As a modern data processing system established around a database and implemented with current computer technology.

- As an abstraction of such a system where all representational aspects are ignored. In this report we call this kind of information system ‘information system in the narrow sense’ abbreviated ISN.

- As a conception of the activities in an organization that are carried out to support and/or ascertain the communication in the organization. This kind of information system is called ‘communication system’ abbreviated CS.

- As a conception of all data handling and communication activities in an organization including systems of the above mentioned kinds as sub-systems. This kind of information system is called ‘information system in the broad sense’ abbreviated ISB.

Common for these different kinds of system is that all seem to be embedded in what we call organizational systems.”

Based on above discussions, we would like to refine the definition of information system into two levels:

1. **Technical systems**: similar to the concept ISN, such systems are “man-made” or “automatic” systems with certain specific functions. We can “make” them, e.g., develop a set of programs and then use them to fulfill specified tasks of information management. The features of such systems are close to those summarized on Page 8;

2. **Business systems**: similar to the concept ISB, such a system means all the information processing activities in an organization or part of it, involving people and various technical systems, and following the rules of the organization leading to certain well-defined objectives. We can not “make” such systems, since in any organization there always has been an IS (ISB), but improve them with our efforts in information system engineerings by replacing some parts of the systems with more advanced technological means.

However, business systems, seen as views of information systems at a higher level, have several different features that are different from those that were regarded as essential of an information system. The examples are:

- **System boundary.** The concept of system boundary is much less important on this level. Naturally, the boundary of an organization may be regarded as the boundary of the business system, but the major attention is paid to the activities within the organization instead of the interactions between the organization and its outside across the boundary;
• **System functions.** The functions of the business system are not some facilities provided to some external entities in its environment, but are the planned and grouped activities happening within the organization in order to support the existence of the organization as a system;

• **System behavior.** Various actions happen in the units of the organization, triggered by various events, but not only by external events from outside of the organization. Even if no stimulus comes from the environment, the system can still be active with various internal events.

Even if today's information system engineering is still mainly aimed at building computerized systems to serve as technical support systems for the information management in enterprises or organizations, there have been more and more requirements to let the systems be connected with each other to form an "overall architecture" [100] rather than to let the systems be isolated. This has identified an important difference between traditional software engineering and modern information system engineering, i.e., "Software engineering applies structured techniques to one project. Information engineering applies structured techniques to the enterprise as a whole or to a large sector of the enterprise" [99]. Therefore, a modern methodology must be defined and used not only for building single technical systems, but also for planning, building and improving a business system as a whole. In this sense, we should do more studies on the characteristics of business systems.

### 1.3 Information system modeling

#### 1.3.1 The role of conceptual modeling

A key factor of success in information system development, like in developments of many other kinds of systems, is the understanding of the system to be developed. To reach this, a conceptual model, i.e., the structured knowledge at some degree of abstraction, is always needed. The importance of conceptual modeling has been recognized in system developments and has resulted in many modeling approaches being proposed during the whole 1980s [142, 143, 70].

It has been proposed [90, 86, 136] that the role of a conceptual model includes the following:

• It serves as a common reference framework used during the system analysis phase to communicate with the future users of the system;

• It serves as a model of reality offering insight into the application domain. In other words, the construction of the conceptual model enables the
system analysts to have a better understanding of the application and the user's needs;

- It serves as a basis upon which the design and implementation of the database can be carried out and against which the design and implementation can be used;

- It provides documentation of the system which can be used during the maintenance phase to facilitate modification and enhancement.

Because of its importance, modeling is used in every system development methodology. A methodology often divides the development into phases and steps along the time dimension, while a system model reflects our view on the system at a specific time point. Therefore, we may have system models at different levels of abstraction in the different phases of the development, e.g., the terms “physical model”, “logical model”, “implementation model”, etc. are often found in literatures. How to express the properties of a system at some levels of abstraction, have been studied for many years by many researchers.

1.3.2 The ontological approaches

It has been claimed that “hundreds, if not thousands” methods for information system development exist [75]. Evaluations of the methods, in a large sense, rely on if there exists a common set of criteria to compare them. There have been some definitions as to what is a “good” model with desirable features [70, 136]. However, in this section we would like to discuss the scopes of modeling methods, i.e., on what abstraction level of information systems a modeling method can be applied.

The basis of our discussion is an “ontological” approach, i.e., “a set of constructs, assumptions and propositions as to what is an information system, from which the design process can be design process can be formalized” [151].

Wand’s model

Building a “ontological foundation” was first explicitly proposed by Yair Wand [151, 150, 152]. His definition of information system is

“An information system is a human-created representation of a real-world system as perceived by somebody, built to deal with information processing functions in organizations.”
"An information system will be considered good if it is a faithful representation of the real world."

To formalize his statement, a real world system is modeled as a triplet $< S, E, L >$ where

- $S$ is a set of legal states of the things that compose the system;
- $E$ is a set of events that happen on the things and may change their states;
- $L$ is a set of laws that include
  - integrity laws that constrain the allowed states of the things;
  - stability laws that map unallowed states resulted from some events to allowed states.

Correspondingly, an information system is also modeled as a triplet $< M, T, P >$ where

- $M$ denotes the states of the information system;
- $T$ denotes the external events;
- $P$ denotes the laws, or processes of the information system.

In this light of systems, Wand declared four requirements for a system to act as an information system:

**Requirement 1**: A mapping exists from the states of the real system to the states of the information system: $R : S \rightarrow M$, such that the mapping $R$ is a homomorphism with respect to the real system law $L$ and the information system law $P$.

**Requirement 2**: For every external event in the real system, an external event exists in the information system that "mirrors" this external event. That is, a mapping $E \rightarrow T$ exists such that if $E = < s_1, s_2 >$, $T = < R(s_1), R(s_2) >$.

**Requirement 3**: The real system is in the environment of the information system that represent it.

**Requirement 4**: Events in the real system trigger their counterparts in the information system in their order of occurrence.

Obviously, Wand's model has ontologically viewed an information system as a man-made technical system. More rigorously speaking, an information system
1.3. Information system modeling

is a reactive system that is always triggered by its environment. This point of view has been applied to evaluate the two famous system models DFD and ER.

For the DFD model, since the three components of the triplet of the IS model can all be found in DFDs, it is claimed that DFDs can be used to represent a proper information system that is a connected set of information system components. However, two conditions are required: 1) there is a path of data flows from any process to the rest of the diagram; 2) every data object is connected to others in the diagram. The first requirement implies that the DFD can not be broken into two or more DFDs, i.e., a DFD at the top level must appear to be a system as a whole. (In my opinion, this condition is often satisfied with the concept of context diagram).

For ER, since no dynamic properties are specified, it is claimed that a system can not be represented in full [152].

The FRISCO model

The FRISCO report [45], though only an unfinished "interim report", is an important attempt to build an ontological foundation for the concepts in the field of information systems. Differently from Wand's approach, the report emphasizes that the perspectives on the concept "information system" rely on the different stand-points of different interest groups, e.g., suppliers of IT products and services versus IT users, and therefore offers a multi-level abstraction hierarchy rooted to the term "system" in terms of the general system theory, see Figure 1.5.

Despite that four different interpretations of the term "information system" are given in the report (see Page 12), the term is still assigned to the perspective of technical systems. Some attention has nevertheless been paid to the concept "organizational system" that is a specialization of the system type "open active systems" (OAS), and some aspects of OSs called "information realization systems" that realizes the information needs of the organizations.

According to the definitions given in the report, the organizational domain (OD) is a realm of human society of the kind usually called organization, company, institution, community, project, enterprise, etc. Correspondingly, an organizational system (OS) is an OAS conceived from an organizational domain. An OS either responds to some events from its environment or has autonomous behavior.

Formally, an OS is modeled as a triplet \(< C_{os}, R_{os}, SP_{os} >\) where

- \(SP_{os}\) defines the systemic properties (states) of organizational systems;
Figure 1.5: The system concepts in the FRISCO approach
1.3. Information system modeling

- $C_{os} = \langle A, P, O, H, T \rangle$ defines the five basic components:
  - $A$: Actors. An actor is a conception of a phenomenon in an organizational domain that is seen as an agent carrying out and controlling the execution of activities in an OS. As a part of the system, an actor may either be a person or a group of persons (human actors), or some artifact actor such as a device or a machine (computer) that has been programmed or set up to perform the duties of being an agent.
  - $P$: Activities. An activity is a connection of a discrete process, with an explicit start and and explicit termination, in the system domain. It is seen as the cause for a certain change of state in the domain;
  - $O$: Operands, being either reagents or knowledge sets;
  - $H$: Events. An event is a change of state in the organizational domain. Potentially, every event may serve as a trigger, say, it causes the proper agents to carry out the execution of the activities. An event may be caused by the environment of an OS, but can also be caused by other sources, e.g., a pre-defined time point is due or an agent decides to let the event happen;
  - $T$: Points in time;

- $R_{os} = \langle Ag, Tr, In, Ou, Ht \rangle$ defines the five basic relationships:
  - $Ag$: Agent role $\langle A, P \rangle$ – an actor performs an activity;
  - $Tr$: Trigger relationship $\langle H, A \rangle$ – an event may trigger an activity;
  - $In$: Input relationship $\langle O, A \rangle$ – an operand is input of an activity;
  - $Ou$: Output relationship $\langle O, A \rangle$ – an operand is output of an activity;
  - $Ht$: Happening relationship $\langle H, T \rangle$ – an event happening at a point in time.

Besides, an important relationship is implied in terms of above relationships:

Knowledge exchange relationship: one actor performs the activity of sending a knowledge set to another actor who performs the activity of receiving the knowledge set.

Based on the framework, when further abstracting the organizational systems to the aspects as information realization system, the concepts information/data and information systems/data handling systems can be defined or explained as:

- An entity is a conception of a specific state in the OD which is seen to be characterized exactly by the set of properties, that is of importance for the function in an OS;
Information in an OS can be expressed as statements about properties of entities;
Data represent information by certain conventions of organized structures of physical phenomena;
A database is a well organized, but variable set of data packages, which during a long period is used in an OS as a means for communication.

- an information/data handling system is a sub-system of an OS that operates on data and possibly store data as a means to support communications and the utilization of information in an organization.

Within such a system, there are some data processors, that are specialized actors, carrying some basic data operations. It was very common that human beings served as data processors before computers were invented, but more and more data handling is taken over by artefact actors (technical devices such as computers with the programs running on it).

It is clear that the FRISCO approach ontologically builds the concept “information system” in a larger context, and thus requires that the information processing activities within an organization must be treated as a whole system. Technical systems, as well as human actors, are only parts of the system and are embedded into the system to fulfill some specific missions of the organization. In the FRISCO approach, Wand’s requirements for an information system model are no longer valid since they can only be imposed in a narrower scope.

Some researchers have noticed such a requirement and have made some efforts to let modeling techniques be applicable in the wider scope. For instance, Essink [47, 48], proposes to build an information system model employing four levels of abstraction:

- the “object system model”;
- the “conceptual information system model”;
- the “data system model”;
- the “implementation model”.

Among these levels, the “object system model” is actually an OS model that specifies the activities and entity types in an organization.

Almost at the same time, Livari [71] also supported the point of view that information systems should be specified at three levels of abstraction:

- the organizational level;
1.3. Information system modeling

- the conceptual/infological level;
- the datalogical/technical level.

These approaches have shown that “more integrated” modeling methods are emerging. The integrated modeling methods will offer the capabilities of specifying systems at the levels that we classified in the last section as the “business system” level and the “technical system” level; the basic components and properties of the systems at the two levels have been revealed by the two ontological studies. On the other hand, in many system development methodologies such an ontological view on information systems still has not been recognized and/or applied.

1.3.3 What is desired for a better conceptual modeling method?

Since “conceptual models have a key role in IS-design” [136], we of course want as many “good” features as possible of a modeling method when we use it. However, what are the desirable features in general?

In [70, 136], some points have been represented explicitly. The two ontological studies we reviewed above have also revealed some desirable properties for a modeling method. Collecting all those opinions and then combining them with my own considerations, the following features may be suggested for of a “good” modeling method:

- Full perspectives. The properties of a system should be specified on not only one aspect. A full perception of the properties can only be obtained from perspectives that reflect all relevant aspects. Therefore, the following perspectives are usually required:
  - data perspective. The types and their structures of data to be managed in a system;
  - process perspective. The processes, i.e., the specific activities with explicit start and end points, executing on some data set to fulfill the tasks of a system;
  - event/behavior perspective. The conditions by which the processes are invoked or triggered;
  - role perspective. The roles of various actors (human beings or technical devices), in carrying out the processes of a system;

- Wide application scope and well-defined abstractions. A complex system can only be specified in a stepwise way, thus abstractions in different scopes and at different levels are needed:
Chapter 1. Basic concepts

- multi-levels of a system, both in the scope of a business system, i.e., the totality of all information management within an organization as a whole, as well as that of the technical system, i.e., an application system with well-defined functions that provide technical support to the operations on data in the processes;

- decomposition, i.e., a well-defined step-wise way to decompose some part of a system (at a higher level) into smaller parts (at a lower level).

• Formal specifications. A system specification is required to be correct and valid so that it contains no error and correctly reflects the system itself. To meet this criterion, the specification has to be given in formal way so that it is:

  - unambiguous. The meaning of the specification is unique so that no misunderstanding may happen;

  - checkable. The properties of the specification can be calculated or verified by mathematical methods or other means;

  - testable. The properties of the specification can be validated by means such as simulation or “model execution”.

• User-friendly description. A system model is built (or formed) for not only technical professionals, but also for common people. In addition, it may be changed from time to time to meet people’s demands or technical progress. Therefore, the system model should be

  - easy to be understood;

  - easy to be modified.
Chapter 2

A survey of modeling methods

In this chapter, we survey "the state of the art" of modeling methods, including those that have been applied in mature system development approaches and some that are still being researched. The investigation will concentrate on the basic conceptual components and features which will be compared with the criteria proposed in the last chapter. Finally, we will give a brief summary of the investigated modeling methods.

2.1 The Structured Analysis approach

The Structured System Analysis and Design approach was developed as early as in the late 1970s [39, 59, 157] by DeMarco, Gane and Yourdon, and has been widely accepted as a system development methodology. We will only study the Structured Analysis part of the whole methodology. The major tools are

- Data flow diagrams;
- Data dictionaries;
- Structured English, decision tables and decision trees;
- Structure charts.

At the analysis phase, the data flow diagrams are mostly used to model a system, so Structures Analysis is often called the DFD modeling approach.
2.1.1 Basic components

In the system model, the following components have all been given a graphical notation: a system:

- **Process.** A process shows a part of the system that transforms a set of inputs to a set of output. It reflects a system activity with specific function(s);

- **Store.** A collection of data staying in a storage.

- **Flow.** A movement of a data package within the system, from one system component to another;

- **External entity.** An individual or organizational unit that is outside the processes of the system to be modeled but interact with them, say, send or receive data through data flows linked to the processes.

With these components, a system or a part of the system is represented as a network of processes, stores and external entities linked by flows.

2.1.2 Leveled data flow diagrams

The advantage of the DFD method is not only that a data flow diagram clearly and graphically represents the activities in a system and their relationships through data flows, but also that it allows a process to be further decomposed into a new DFD with a set of smaller processes and other elements. By the way, a system can be specified in a step-wise manner and as a hierarchical structure. Finally, when a process is considered as having reached to a detail level where no further decomposition is needed, a “process logic” can be defined in forms of structured english, decision table, decision tree, etc. that illustrate the procedures or rules for executing the process.

Since DFD is not a formal model, there is no widely agreed regulations to give restrictions on the DFDs. Some authors have suggested rules, but that are often in conflict with other opinion. For instance, one standard defined for “good” DFDs in [66] is obviously different from the proposal of [156]. Moreover, DFDs are ambiguously. This has been pointed out in [156].

However, there is a common opinion: the DFD model is a functional approach, so each process represents a function that receives some inputs and then produces some outputs. When a process is decomposed into a set of sub-processes, the sub-processes are grouped around the higher level process, and are just cooperating to fulfill the higher-level function. This view on DFDs has resulted in the “context diagram” [59, 28, 81, 156] that regards the whole system as a
2.1. The Structured Analysis approach

process which receives and sends all inputs and outputs to and from the system. Such a view of a system can be illustrate with a leveled DFD structure which is rooted to the context diagram, see Figure 2.1.

![Diagram of context diagram with levels 0, 1, and 2]

Figure 2.1: The leveled DFD structure

A context diagram has in fact determined the boundary of a system. Every created activity (process) of the system is seen as the result of a stimulus by the arrival of a data flow across some boundary. On the other hand, the task of a process is only to produce and send out the requested outputs that correspond to the stimulus. Consequently, if no external data flow arrives, then the system will remain in a static states. Therefore, The DFD method is still basically a modeling tool for specifying reactive systems.

2.1.3 Evaluation

The advantages and disadvantages of the Structured Analysis approach can be summarized according to the criteria suggested in section 1.3.3:

- Perspectives: On the process perspective, DFDs are very strong;
  On the data perspective, DFDs have provided the modeling component stores and the modern structured analysis has connected the data stores with the data entities in the ER model [156];
  Meanwhile, on the event/behavior perspective, DFDs are weak since neither time condition nor any other logical expression is given, with the exception of the arrivals of external data flows;
On the role perspective, DFDs define the roles of the external entities as the agents to interact with a system. However, within the boundary of the system, there is seldom explicit definitions about the roles of the system parts, since the concept of actor is not given in the DFD method. Even if some parts of the system can be assigned to become a computerized system (by defining a new boundary between the parts of man's work and that of computer's work) [59, 28], there are still no explicit definitions about the roles of the different parts within the "computerized system".

- Application scopes and abstraction capability: The most important advantage of the Structured Analysis is its step-by-step evolution of the analysis. Even though no strict distinguishment is defined between the business system and technical systems, the DFDs have provided a way to start specifying a systems activities from a higher level where peoples' participations are viewed as parts of the system, and gradually push the specification to the detailed level where computer support is possible. Nevertheless, if a "context diagram" is imposed on the top of a DFD hierarchy, then it is difficult to only use the DFDs included in the DFD tree which is rooted from the context diagram to specify the information processing activities in a whole organization.

- Formal specification: On this aspect, the Structured Analysis is quite weak since few formal concepts are defined for DFDs besides the basic components. Therefore, it is easy, especially at a higher level DFD, to get ambiguous answers when asking for the properties of the system elements.

- User-friendly description: Structured Analysis has very user-friendly description. The basic components are simple and have been given simple graphical notations.

Finally we would like to mention that the Structured Analysis approach is highly in concordance with the structured programming paradigm. This is not strange since both paradigms appeared in 1970's as a means to solve the so called software crisis.

2.2 The Entity-Relationship Model

The entity-relationship model, since it was first proposed by Chen [22], has also become a widely-accepted model to describe the data types to be managed in information systems.

2.2.1 Basic components

In Chen's original paper, the basic components are
2.2. The Entity-Relationship Model

- **Entities.** An *entity* is a "thing" which can be distinctly identified. Entities can be classified into entity sets;

- **Relationships.** A *relationship* is an association among entities. Like entities, relationships can also be classified into relationship sets;

- **Attributes and data values.** A *value* is used to observe and measure the information about an entity or relationship. Values are grouped into value sets by their types. An *attribute* is a function which maps from an entity set or a relationship set to a value set; thus the property of an entity or a relationship on a special aspect can be expressed by an attribute-value pair.

With these components and their graphical symbols, a the results of system analysis can be expressed by drawing an ERD (entity-relationship diagram) that contains a set of entity sets, relationship sets among the entities, and the attributes of the entity sets and relationship sets.

The ER model has been studied in numerous variants [113, 140, 65, 145].

2.2.2 Evaluation

- Perspectives: The ER model only provides the data perspective on a system, thus it can not fully describe a system. Therefore, the ER model should always be used in combination with other models such as DFDs.

- Application scope and abstraction capability: There is no limitation to the scope in which an ERD can represent the entity types and relationship types. One may also merge several ERDs into one. The ER model can be applied to describe the data types of the whole organization.

  The term *data abstraction* is often used to describe the relations or constructors between data (or entity) types, such as *generalization/specialization, aggregation, grouping,* etc. In the sense of the definition of abstraction given in this thesis, The ER model has very limited abstraction capabilities. The most often seen abstraction is that an ERM is drawn with only entity types and relationship types without attributes. Those are to be specified at a more detailed level. Aimed at higher level abstractions, there are only a few approaches that attempt to "cluster" entity and relationship types into higher level types [141, 30]. The approaches have not been widely accepted.

- Formal specification: The ER model uses *set theory* as the foundation of its concepts, so it has been possible to formulate a formal mode. Most of its variants are also formal models, e.g. that given in [5].

- User-friendly description: One of the main purpose of proposing the ER model is to let the data types in application are described independently
of any concrete implementation technique. Therefore, the model has been described as "one of the earliest semantic data models" [69].

2.3 More semantic data models

After the early approaches to semantic data modeling, including the ER model in the late 1970s, the research on the topic continued into the 1980s. More semantic data models, such as PM [5], SDM [64], FDM [132], TAXIS [49], SHM+ [15], IFO [123], etc., were proposed. Overviews are given in [69] and [116].

Generally speaking, semantic data models provide an abstract view on the objects that carry information in an information system. This is called "logical independence" in the database field, since the details of physical implementation should be hidden within a DBMS. Consequently, most semantic data models are not restricted by the structural constraints that are imposed on conventional data models, e.g., the 1NF constraint for the relational databases. This results in rich type (class) structures in most semantic data models.

2.3.1 Basic components and concepts

In Hull and King's treatise [69] a generic semantic model (GSM) is given with a simple example for the schema of the World Traveler data base (Figure 2.2). The components and concepts of GSM are:

- **Primitive types.** The data types in GSM are classified into two kinds: the printable data types, e.g., PNAME, BMAME, ...etc., that are used to specify some visible values, and the abstract types that represent some entities in the world, e.g., the PERSONs or BUSINESSes. The data of abstract types are kept known within a system by internal identifiers that are neither visible to users nor printable.

- **Constructed types built by means of abstraction.** The most often used constructors for building abstractions are
  
  - **specialization.** The Specialization/Generalization relationship is also called IS-A relationship. It specifies some types as sub-types of other types. A sub-type inherits the properties of its super-type. In the example, the types TOURIST, BUSINESS TRAVELER and LINGUIST are all constructed in this way;

  - **aggregation.** It generates new types by compositing several types together. Each of the participating types is called a component,
2.3. More semantic data models

or a part of the constructed type. In the example, the data type ADDRESS is constructed in this way;

- grouping (or association). It constructs a new type, a value of which is a set whose elements are of an existing type. In the example, the types LANGUAGES and DESTINATIONS are constructed in this way.

- Attributes. Besides the construction relationships among the types, another kind of relationship called attribute can also be defined. An attribute of a type is a function or a map from the domain of the type to that of another type; it defines a property of the type. In the example, the attribute HAS-NUMBER, SPEAKS, ENJOYS, etc., define some properties of the abstract types.

GSM represents the common features of semantic data models. However, each individual model may have its own special features. It is worth mentioning that the relationships between instances of types may be defined in different way. We see in Figure 2.2 that a relationship is defined by a two-way attribute (an attribute and its inverse). In the E-R model, a relationship is represented as an explicit type. The definition of relationship types provides the possibility of specifying such relationships among the instances of more than two types as well as that of defining attributes of such relationship types. This is an
important reason for that the E-R style data models has gained high popularity in the database community.

Like the E-R model, a GSM system model has the form of a database schema in which a set of types (or classes) and the relationship among them are defined.

2.3.2 Evaluation

- Perspectives: Generally speaking, semantic data models concentrate on the data perspectives of information systems. Since the way that they specify the objects of a system is close to the way that people observe a reality, rather than the way that is relevant to a DBMS implementation, semantic data models provide rich structural components for complex data types.

Most semantic data models have few concepts for the dynamic aspects of a system, except the generic manipulation operations on the data objects. Some models, e.g., TAXIS and SHM+, allow a DB schema to contain the definitions for specific transactions on the data types or classes. Nevertheless, the transactions only consist of low level operations on the data objects, so they can hardly be viewed as descriptions of the processes of a system.

- Application scope and abstraction capability: In general similar to those of the E-R model.

- Formal specification: Most semantic data models are formal models expressed in various forms.

- User-friendly description: Most semantic data models allow the user to create conceptual schema whose components are understandable to common users. Because the formal specifications may be difficult to understand for people without proper formal training, the models are usually given with some graphic notations which may be easily understood.

Unlike the three conventional data models, the semantic data models have not yet been widely applied commercially with available DBMSs. We can list some reasons: 1) although there has been a common set of concepts for semantic data models, no “standard” model exists; 2) there is no widely accepted data language such as SQL for the relational data model; 3) there have been many databases built by means of conventional data models and the enterprises are unlikely to transform the data in these databases into the forms required by a new type of DMBS. Therefore, in information system engineering, the semantic data models are mostly used only as a “conceptual modeling tool” during the design phase.
2.4 The Jackson System Development

The Jackson System Development method was developed by Michael Jackson and his colleague John Cameron during the early 1980s [78, 21]. Based on the philosophy that a model of the UoD is more stable than the functions that a system performs, the basic principle in JSD is to let the system simulate the relevant parts of the real world outside the system and to provide system functions that get inputs and generate outputs as responses to the behavior of the simulation. JSD has also been widely applied to build various systems, especially real-time systems.

2.4.1 Basic components and concepts

A brief description is selected from the introductions of [28, 104]. The system model, in the sense of our definition, is formed during the steps of entity/action and entity structure, initial model, interactive functions and information function [104] with the following components:

- **Entity.** Different from the definition in the ER model, the concept *entity* in JSD is something existing in the real world and perform or suffer actions in a time order. It is outside the system which is to be built, but its actions must be reflected within the system.

- **Action.** An *action* is an atomic event, occurring at some point of time, and considered to be instantaneous. Since actions are always carried out with some entities at special moments in their lifetimes, an action must take place in the real world outside the system.

- **Entity structures.** The *entity structure* of an entity, given in a form of diagram, shows an entity's entire lifetime consisting of actions that it performs or is performed upon it, with the execution directions *sequence*, *iteration*, or *selection*.

- **Process** A *process*, often called *sequential process*, is an execution sequence. An entity structure itself can be viewed as a process in the real world, while a process can exist within the system as the *simulation* of the real world process.

- **Function.** A function is a service of the system, which means the retention of data about the subject matter of the system, and the provision of outputs which select and summarize these data in the form required by the user.

- **State vector.** The *state vector* of a process consists of its *local variables* that are assigned to some attributes of the process during its execution.
• **Data stream communication.** One process may communicate with another process by operations on a named *data stream*. The operations on a data stream are *write* and *read*.

• **State vector communication.** The values in the state vector of a process can only be changed by the execution of the process itself, but may be inspected by other processes.

### 2.4.2 System modeling

The system model is built by the steps

• Specify entities, their actions, and the entity structure. All can be given in *structure diagrams*. The specification is called *the model of the world* in terms of JSD. An example is given in Figure 2.3 where the structure diagrams for two entities are shown;

![Structure Diagrams](image)

*Figure 2.3: The structure diagrams for two entities*

• Define a set of sequential processes that model the real world within the system, i.e., in the system, the real world is simulated. The simulation is described by defining the "real world processes" and their corresponding "sequential processes within the system" with the data stream communications to let the system processes keep the same paces with their counterparts. The system processes are specified by both graphical notations and *structure text*;

• Add system function processes and build necessary communications between them and the previously defined system processes. At the end of the step, the system is defined as a simulation of some entities in its environment with certain specified function. The example given in Figure 2.3 is now developed into the system structure shown in Figure 2.4.
2.4.3 Evaluation

- Perspectives: Only process and event/behavior perspectives, i.e., the properties of the dynamic aspects, are provided. The event/behavior perspective are reached from the simulation mechanism whereas the process perspective is provided by means of the "function processes". Data are only used for the purpose of communication between processes. The database issue is not treated at all. Even if the further implementation steps may divide the system into some programs, they are viewed as being within ONE technical system. Consequently, no actor concept is needed.

- Application scope and abstraction capability: The modeling is limited to the scope of technical systems, particularly real-time systems with strong time-dimension requirements. Meanwhile, there is no abstraction mechanism in JSD. All specifications must be given at a detailed level.

- Formal specification: since the JSD is essentially an application of the communicating sequential processes approach [67], a system model can be transformed to a set of formal expressions.

- User-friendly description: For a small system, the system model can be easily understood; however, when the size of systems grows, one may need the help of competent persons to get insight in the system model because of the lack of abstraction facilities.
2.5 SSADM

SSADM (Structured Systems Analysis And Design Method) [29, 43] is a system development methodology which was initially developed by Learmonth and Burchett Management Systems, LBMS, and was finally accepted by the UK government's Central Computer and Telecommunications Agency as the UK government's standard method in system analysis and design.

2.5.1 Basic components and concepts

SSADM uses DFDs and the ER model as its basic modeling tools, but also defines the cross reference between the two model. It contains the following components and concepts:

- The components of DFD;
- The components of the ER model;
- Data store/Entity cross reference. The cross references between a data store and some entity/relationship types describe what kinds of data are to be kept in the data store;
- Entity/function matrix. The matrix lists entity/relationship names in rows and function (process) names in columns, and let the filled letters ("I", "R", "M", "D") indicate the operations that a function performs on entities.
- Entity life history. The history of an entity represents the sequence of functions carrying on it during its life time.

2.5.2 System modeling

Although it is required in different phases to create both a physical model and a logical model of a system, the system is always modeled as a set of DFDs, an ER diagram, and the cross references between them (Figure 2.5).

2.5.3 Evaluation

- Perspectives: Both data and process perspectives and supported and are integrated with cross references between the two perspectives.

With respect to the event/behavior perspective, SSADM is weak because no special components or concepts are provided, except that the arrivals of data flows imply some events.
Figure 2.5: The system model in SSADM

With respect to the role perspective, SSADM may specify that some departments or persons play the role of agents in carrying out the processes in a "physical model" of the "current system". Nevertheless, once the physical model is transformed to a "logical model" for the "future system", then the roles are no longer specified. Moreover, when the design phase is reached, only those parts of the processes that are to be computerized are specified; consequently, the roles of the programs after the system has been designed do not need to be specified either.

- Application scope and abstraction capability: Although the ER model has a wider application scope than that of structured analysis, since in SSADM every element of the ER diagram must be connected to at least one process, the scope of the modeling capability is in fact bounded within that of the DFDs.

On the scope of the DFDs, SSADM start the structured analysis by drawing a special DFD on which only several dynamic entities (the external entities in terms of the original DFD method) and the data flows among them are drawn, then some of the entities (often with the names of some departments in an organization) are exploded into the top level processes while other entities are viewed as external entities outside the boundary of the system to be developed. This is actually a similar mechanism as that of "context diagram". Therefore, the scope is also similar to that of Structured Analysis.

- Formal specification: The same problem with that of Structured Analysis.

- User-friendly description: Good, since both DFDs and the ER model are quite user-friendly.
2.6 IE

IE (Information Engineering), developed by James Martin together with many of his colleagues [103, 93, 92, 99, 101, 102], is also a widely applied methodology for analysis and design of information systems.

The key feature of the methodology, if not unique among various development methodologies, is that IE aims at not only building individual IT systems, but also making a strategic plan of information resources and relating the plan with the strategic business plan for a whole enterprise or across a major sector of the enterprise[100]. Such a goal requires an overall architecture to be built based on the business activities of the enterprise, rather than an isolated information system. Therefore,

"IE has sometimes been described as an organization-wide set of automated disciplines for getting the right information to the right people at the right time."[99]

2.6.1 Basic components and concepts

Similar to SSADM, IE use multi-modeling tools to provide enterprise model, data model, process model, etc., and the cross references among different models. The difference is that in IE the modeling is started and carried out at the enterprise-wide level. The models may be independent of any specific application. The characteristics of IE modeling are further described as follows:

- **Data models.** The basic premise of IE is that data lies at the center of modern information systems methodology, because "basic data types are stable whereas procedures tend to change". Therefore, data models are considered "particularly important". IE uses the ER model (and therefore contains all its components), but it is an ERM variant with generalization/specialization abstraction.

- **Business areas and business functions.** IE does not start at a specific application, but at the overall business structure of the enterprise. Therefore, the business functions and their areas must be specified.

- **Process models.** In early version of IE (see [28]) no process component but procedure is used. The revised IE [92, 101] includes a process modeling that is partly similar to the DFD method, but many process models instead of only one may be built. The the construct context diagram is never used.
2.6.2 Systems modeling

IE is used enterprise-wide to build large and complex information management structures. IE has four levels that indicate the development phases from the top level to the detailed construction level. The four phases are called: 1) Information Strategy Planning; 2) Business Area Analysis; 3) System Design; 4) Construction. Specific IE modeling techniques are mainly used during the first two phases.

In the first phase (ISP), an overview model of the enterprise is created, that identifies the organizational units, locations, business functions, entity types, and the relationships of them.

The entity types are refined so that an initial entity-relationship diagram is formed. The major functions are refined as well, to let more specific business functions be identified. Afterwards, several matrices are built, especially the function/entity matrix that let the functions be clustered into groups that serve as the sub-systems of the enterprise. A function group, together with the relevant entity types, is called a business area (BA).

A BA includes one or more business functions. A business function is realized by a group of specific and cooperative activities, called processes, each of which is executed periodically under specified conditions to fulfill a particular aspect of the function. Furtherly, a process may be composed into a set of smaller processes until a detailed level. Such a functional decomposition is shown in Figure 2.6 that reveals an overall structure of the business activities of an enterprise.

Work in the second phase (BAA) establishes a detailed framework for building an information-based enterprise. One business area at a time is analyzed in detail by means of

- data modeling that starts from the ERM that is created during the previous phase, but at the BAA phase all entities are regarded as data records to be stored. They are "normalized" to the fourth or third "normal form". Consequently, the so called "logical data model" becomes a relation-like data model;

- process modeling. A top level process is one of the specific activities of a business function; it does not necessarily have to be specified with inputs and outputs. Like the processes in the DFD method, a process may be decomposed, but the decomposition is expressed by three diagrams: a compose-of diagram which shows how many sub-processes (sub-activities) that are included in the process; a process-dependency diagram which shows how the sub-processes are related to each other and their execution sequences, and a process data flow diagram which shows the inputs and outputs of the sub-processes and the data flows among them.
When the BAA ends, a foundation for the systems design has been prepared. The process logic of a bottom level process can be expressed by an action diagram and tested, validated or even practically coded with the fourth-generation languages (4GL) [98].

2.6.3 Evaluation

- Perspectives: Like in SSADM, both the data perspective and the process perspective are fully supported and integrated, but it is relatively weak on the event/behavior perspective, so IE is not suited to the development of systems with a strong time dimension.

With respect to the role perspective, it has been noticed that technical systems can be formed as clusters of processes with explicit action diagrams, and becomes sub-systems (real "technical information systems") in the enterprise [92, 101].

- Application scope and abstraction capability: The most important advantage of IE is that it supports the organizing of the information management starting from the needs of the business functions of the enterprise, rather than starting from single application systems. Hence, IE provides abstraction hierarchies from high-level business management to detailed technical operational level. This makes IE particularly powerful for building complex and large information systems.

- Formal specification: For the data modeling, the formal methods for "normalization" are applied; for the process modeling, even though more
information is provided than in DFDs e.g., the process-dependency relations, it is still basically an informal model.

- User-friendly description: The concepts of individual models are quite easily understood; however, since IE is a complex compound of models and engineering disciplines, it still needs well-trained and skilled experts to make correct and best use of the method.

2.7 BNM and REMORA

In the previous sections we have presented some modeling methods that permit us to build sub-models of different aspects of a system when multi-perspectives are needed. Now we review two modeling methods, proposed by Rolland et. al. [119] and Sølvberg&Kung [90] respectively, that attempt to integrate several perspectives into one model.

2.7.1 Basic components

The BNM integrates the *Phenomenon Model* which is a variant of ER, with a *conditional petri net model*. Its basic components include

- **Entity and relationship classes.**
- **Token.** A token has the same meaning with that in the Petri Net Model and may be an instance of an entity or relationship class (denoted by “∈”), or the whole class (denoted by “=”).
- **Place and transition.** They have the same meaning as the Petri Net Model, but a pair of logical expressions — called pre-condition and post-condition — is assigned to each transition to show the condition and the result of the firing of a transition.

REMORA explicitly defines the following concepts:

- **Object** – an entity, i.e., a concrete or abstract component of the reality that is being illustrated;
- **Operation** – an action that can be executed at a given time and modifies the state of objects;
- **Event** – something that happens at a given time and has consequences for the system behavior.
Chapter 2. A survey of modeling methods

The relationships between the concepts are:

- An operation **Modify** the states of objects;
- Events **TRIGGER** operations;
- A particular state change of objects can **ASCERTAIN** an event.

### 2.7.2 The conceptual modeling of systems

Both of the modeling methods support the creation of a *conceptual model* of a real-world system by linking their components. Nevertheless, the models are both particularly *behavior oriented*. Two examples are given in Figure 2.7 to show the system models with BNM and the REMORA method.

For BNM (Figure 2.7.a) we see that the arrival of a token to a place which has a solid line connecting it to a transition (not the place with dotted line since the token within it is always available and is never removed after the firing) will trigger the evaluation of the pre-condition of the transition. If the condition is true, then the transition is fired and the triggering token is removed. After the firing, the post condition of the transition becomes true with possibly produced new tokens in other places. Then other transitions may possibly be triggered and/or fired. The system activities are illustrated by the firings and the corresponding movements of the tokens.

For REMORA (Figure 2.7.b) we see that the system is triggered by the arrival of a reservation demand (*OB1*). When the event happens, some conditions should be tested (e.g., *C1*: the reservation demand can be satisfied), then some suitable operations (e.g., *OP1*: reservation state creation; *OP2*: reservation room creation; *OP3*: reservation creation; etc.) are executed and result in more states changed for other objects, so that more events are ascertained and more operations may be further executed.

### 2.7.3 Evaluation

- Perspectives: Both the two modeling approaches support integration of the data perspective and the behavior perspective, so that the data to be manipulated and the operations that are handling those data can be specified simultaneously in one diagram.

However, on the case of process perspective, BNM and REMORA are both weak because they can not indicate what operations (transitions) belong to a special activity which realize a specific task. The problem appears when more complex systems are to be specified. The creators of
a) An example in BNM: an order handling system

b) Ex example in REMORA: a room reservation system

Figure 2.7: System models built with BNM and REMORA
BNM have been aware of the problem, and further proposed to use BNM only at the bottom level of a process model [86].

The role perspective is not supported in the two modelings.

- Application scope and abstraction capability: Both modeling approaches specify a system at a fully detailed level, thus their modeling capability have actually been bounded within the scope of technical systems or the real world around a technical system. It is also difficult to abstract the specification of a system at multi-levels.

- Formal specification: The two modeling approaches are both formal, so that the entity (object) concept can be expressed by means of predicates whereas the operations can be expressed by rules.

- User-friendly description: It will be hard for a common user to understand a system specification given by the two modeling approaches, especially when the system is large.

## 2.8 Object-oriented modeling

It is not strange that object-oriented conceptual modeling has become a hot topic in the information system area [53]. When the first object-oriented programming language SIMULA was developed [35] in 1960s, the concept of object had been used for the purpose of simulation. Even after many new OO languages have been developed [56], the general characteristic of object-oriented programming is still “A program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world” [94]. From a technical point of view, the evolution of the object oriented languages and consequently the object oriented systems and system development environments built with the OO paradigm, have provided new types of technical platforms that create the needs and possibilities for object-oriented conceptual models of information systems [24, 25].

On the other hand, it is believed that the OO paradigm has provided a uniform way of dealing with the static and dynamic phenomena, so that the “artificial boundary between active and passive objects and between persistent and transient objects” is overcome [51]. Moreover, the formal theory about abstract data types (ADTs) [46] can be used as the basis for formalization of the concepts in the OO modeling. Many approaches have been proposed, based on the theory of algebraic specifications and temporal logic [2, 3, 130, 51].
2.8. Object-oriented modeling

2.8.1 Basic concepts and system modeling

The basic concepts of object oriented modeling are similar to those found in most OO languages:

- **Object.** An *object* is an "entity" which has an unique and unchangeable identifier and a local state consisting of a collection of attributes with assignable values. The state can only be manipulated with a set of *operations* (also called *methods*, or *actions*) defined on the object. The value of the state can only be accessed by sending a *message* to the object to call on one of its operations. The details of the operations (*procedures*, or the *implementations* of the methods) may not be known either, except through their interfaces. This property is called *encapsulation*. The happening of an operation being triggered by receiving a message, is called an *event*.

- **Process.** The *process* of an object, also called the object's *life cycle*, is the trace of the events during the existence time of the object.

- **Class.** A set of objects that share the same definitions of attributes and operations compose an *object class*. A subset of a class, called *subclass*, may have its special attribute and operation definitions, but still share all definitions of its super class. This is called property *inheritance*.

With these concepts, a system model is composed with a set of objects. Each object has its local state, and interacts with each other objects by sending and receiving messages. Such a system model is also called *an interacting objects society*.

2.8.2 Evaluation

It is hard to say what is exactly OO modeling. Many authors have presented their modeling approaches, more or less, with the concepts of OO paradigm. Many views have been presented, but no concensus has so far formed on methodological issues. The evaluation can only be given to the generally agreed framework rather than a specific method.

- **Perspectives:** The object oriented modeling has provided an integrated model to deal with the data aspects and behavior aspects of the objects. The system may be described as a collection of interacting objects; it is often parallel to man's perception of the real world.

For the process perspective (not in the sense of *object life cycle*), a process in a system is considered as "the inter-object triggering in terms of the order of events on multiple objects", so "a process is not associated with
a single object, but a collection of objects" [118]. In other words, a process is a sequence of actions carried out by the collection of objects that triggered the actions by sending messages among the objects.

With respect to the role perspective, the concepts agent and role are adopted in many OO modeling approaches to let agents be "assigned specific Roles to perform certain functions and servises" [74].

- Application scope and abstraction capability: Conceptually, there is no limitation for the scope of an object-oriented model since the paradigm is intended to describe the real world. However, since an OO model always describes a system as a flat structure on which the properties and actions of the objects and their associations are specified in detail, this way can hardly be used to describe and analyze complex business activities at a high level. In other words, The OO method, when used to do business analysis, lacks the capability to specify system activities in a step-wise manner.

- Formal specification: The OO method can be expressed in very formal way. In fact, there have been several formal specification languages developed, e.g., OBLOG and TROLL, with algebraic semantics [2, 117].

- User-friendly description: Although the formal languages might be difficult for common users, the modeling concepts of the object oriented modeling are quite user-friendly.

2.9 Comments on the existing modeling methods

Having investigated some existing modeling methods, we conclude as follows:

- Many modeling methods have been developed, with some desired features, and are effectively used in information system engineering;

- On the other hand, most conceptual models we investigated do not have ALL of the desired properties that we illustrated in Section 1.3.3. These properties are needed not only for building high quality systems and avoiding errors in the process of analysis and design of systems, but also for meeting the requirements of building more and more complex modern information systems on the newer generation of technical platforms;

- It seems difficult to reach the two desired properties of multiple abstraction levels and formal specification simultaneously. For instance, the Structured Analysis and SSADM support decomposition of processes, nevertheless they are both informal; On the other side, although BNM, REMORA and the object-oriented approaches have solid mathematical
2.9. Comments on the existing modeling methods

foundations, they can only specify a system on a flat structure. Since the step-by-step analysis is very helpful in developing systems in large, a good modeling method should build a bridge between the two aspects.

- Except Information Engineering, few methods start to build a system model from the organization's business activities instead of from the requirements to a technical system. Consequently, little attention has been paid to how to embed technical systems into the business activities, although this is actually the main purpose of information system developments.

- In most system development methodologies, the terms design and implementation are still regarded as some kind of programming process, i.e., to design the module structure of some systems, then code them with some language like COBOL, PASCAL, ADA, C or 4GL. The existence of the software industry and many software products sold in the market have often been ignored by methodology developers. Therefore, few modeling methods provide conceptual components for describing the properties of software products that are available in the market place.

The above statements have revealed that we still have much room for improvement of the modeling methods. In the following chapters, we will make efforts to build a new modeling framework that may achieve some progress toward the goal given in Section 1.3.3.
Part II

The new modeling approach
In this part the new modeling approach is presented.

In Chapter 3 the modeling components are introduced. The COMIS (COncceptual Model for Information Systems) framework includes three sub-models: the extended E-R mode ONE-R (Our New E-R model), the Process Port Model (PPM) and the Actor Model (AM) as well as the cross references among the modeling components.

In Chapter 4 the consistency check methods are presented. Since COMIS is an integrated modeling framework with several sub-models, the definitions of consistency and the corresponding verification & check methods must be designed to fit on the characteristics of it. Basically, we will use logical methods and the STD method to deal with different aspects to the specification of a system.

In Chapter 5, a CASE tool is introduced, which supports many of the modeling concepts given in this thesis.
Chapter 3

COMIS – an integrated system modeling approach

In this chapter an integrated system modeling approach, COMIS, is presented. The reasons to claim that our approach is "integrated" are:

- A system model is established, which supports a multi-perspective, i.e., systems may be specified with all of their data, process and behavior aspects. Several sub-models, as well as cross-references among them, are defined for the purpose.

- We notice that the scope of the information processing activities within an enterprise should be bounded not only by technical means. The modeling approach should therefore be able to specify business systems and associated information management. On the other hand, we are aware of that the major objective of information system engineering is to build computerized systems which support the mission of the business systems. Therefore, our modeling approach must also be able to specify the technical systems. COMIS has enough conceptual components to meet the requirements from both sides. Moreover, the specifications of business systems and technical systems are put in an integrated modeling framework.

The conceptual framework is introduced in the first section to give an overall structure of COMIS. In the later sections, the sub-models will be presented.

3.1 The modeling framework

In order to satisfy the criteria for conceptual modeling of information systems that we defined in Chapter 1, it must be possible to use our modeling approach
within the scope of business systems development. For a business system, the two major aspects of business activities and data used and processed in these activities, must be specified so that a thorough understanding of the information management within the business system can be obtained.

In COMIS, the specifications of the data processed in a business system are grouped into scenarios while the specifications for the business activities are grouped into business functions. A business system model is given in Figure 3.1 where:

- A scenario represents a piece of reality whose information is to be handled in the business system. The phenomena in the reality are expressed by a set of entity types and relationship types.

- A business function is a group of activities that cooperate to support one aspect of the business system for its existence as a system. It is ongoing and continuous, i.e., a business function is always active or may be activated during the entire life of the business system.

  Within an activity group, any specified activity that supports the business function is called a process and may be executed repeatedly. We can also decompose a business function into a set of processes having various triggering conditions with definable starting and ending points, and that can be described in terms of inputs and outputs.

The business system model is independent of any particular technical solution. However, since the major objective of information system engineering is to
build computerized technical systems to support the information processing work of a business system, our model must also be able to deal with the technical systems.

In Figure 3.2, a model for computerized technical systems is shown. The model is conformable with the ANSI-SPARC architecture [31] for data schemas because our modeling approach is strongly oriented towards database centered systems. Besides, since the object-oriented approaches have influenced software techniques strongly, we have also introduced the concept of object into our model, where the database contains passive objects (entities) and the application system contains active objects (actors).

![Diagram of technical systems model]

Figure 3.2: A technical systems model

Considering an enterprise as a system with a composite (or hierarchical) structure, a business system is at a higher level than the technical systems. A business system may have several databases and application systems as the technical means to support its activities. Can our modeling approach show the relationships between a business system and the technical systems underlying it? Can the descriptions of the two kinds of systems be put together into an integrated modeling framework? The information system model we illustrate in Figure 3.3 provides an answer:

- A business system is represented by its business functions and scenarios of data. Each business function consists of a set of processes triggered with specified conditions repeatedly; each process can be further decomposed
Figure 3.3: An integrated information system modeling framework
into a set of sub-processes; the decompositions can be further done until a bottom level is reached, at which the processes (primitive processes) do not need to be further decomposed any more and may be specified with a procedural or logical language directly. On the other hand, a scenario consists of a set of entity types and relationship types that represent a piece of the reality whose information is to be processed in the business system.

- The technical systems within the business system consist of database(s) and application system(s). A database stores some data of the business system. An application system has one or more actors (programs) with some specified functions which can be invoked to support the processes of the business system.

- Because of the developments of semantic data models, object-oriented data models and new DBMSs directly based on the new data models, we assume that the conceptual schemas of the databases may be written in terms of the new data models. On the other side, the data model by which we describe the data types to be managed within the business system is built by similar conceptual components too. Therefore, the conceptual schema for any database is just one or more scenarios of the business system.

- The actors of an application system are used to support some (not all) primitive processes. When one function of an actor is called, it will have the same inputs and outputs as those of the process this function supports. Thus, the behavior of a new actor can by specified by grouping a set of primitive processes. When the actor is created or purchased (i.e., a program with the requested functionalities is designed or copied), it will be installed and executed to support the processes.

- An application system works on some parts of the database(s). Since the conceptual schemas of the databases are defined by the scenarios, we may also define some external scenarios that copy or map (by means of abstraction or derivation) some contents of the scenarios to the application systems.

Within this framework, three major models are used to specify a business system and its supporting technical systems:

- An extended entity-relationship model called ONE-R (Our New Entity-Relationship model);
- Process port model;
- Actor model.
3.2 The Extended Entity-Relationship Model (ONE-R)

3.2.1 Introduction

ONE-R is an extended entity-relationship model (Our New Entity-Relationship Model). It is developed on the basis of Phenomenon Model (PM) [4, 5] which itself is a variant of the ER model.

Chen’s ER model [22] is the first semantic data model [69] which has had a wide acceptance. It motivated the work on a number of extensions, e.g., those presented in [113, 140, 65, 145]. In the 1980s, more work has been done on semantic data models such as TAXIS [49], SDM [64], FDM [132], that supported complex data structures and various abstractions for the relations between data types. The characteristics of the models have been summarized in [69] and [116]. Meanwhile, Object-oriented data models such as ORIGON [76], POSTGRES [120], BIM-Probe [11, 38] and O₂ [52, 17] have also appeared, and made it possible to define both structural and operational aspects of data objects.

The goal of ONE-R is to provide a modeling tool to specify the data objects within information systems at the conceptual level, with the features of new technical platforms fully considered. We take PM as the basis since PM has provided important concepts such as class hierarchy, set valued attributes, and structural constraints on the relations of the ONE-R constructs. Then more components are added:

- ONE-R is both an extended ER model and an object-oriented model.
- ONE-R has simple and diagrammatic specification formats, while a formal structure is defined too.
- ONE-R has an algebra with a set of well-defined operators, then on the basis of the the operations, an SQL style Semantic Data Language (SDL) is defined with DDL, query and DML components.

The work to define ONE-R has been influenced by many research approaches:

- Previous proposals of ER style models, including extended E-R models and their algebras and languages [23, 148, 33], have all contributed to the improvements on both structural and operational aspects of E-R. Markowitz and Raz’s concepts reshaped relation [97] and its algebra has helped us to define the expression form of entity and relationship types. Parent and Spaccapietra’s ERC+ model and its algebraic operator set
[112, 113, 115] provides operations on the entity sets directly. A special operator, relationship join, derives a new entity type from an existing entity type in relationship with other entity types, and groups all the attributes of the other entity types and the relationship itself into a multi-valued attribute of the new entity type. This treatment gives the algebraic operations in ERC+ a strong semantics, and provides a good foundation for us to design the ONE-R algebra and SDL.

- The nested relation models and the complex data models, starting during the early 1980s are important sources for the proposal of ONE-R. Schek, Roth, Özsoyoglu and others developed the NF2 relation models that defined the concept of relations with relation value attributes or set-valued attributes. They also developed the corresponding relational algebras and languages (e.g., SQL/NF) [129, 91, 57]. These algebras or languages give us good templates for the algebraic operators defined on entity and relationship types in the form of “reshaped relations”. Abiteboul, Beeri, and Hull et. al have discussed various structures for complex data type and the manipulations on them [68, 124, 123, 69, 1]. These results are helpful to build the type hierarchy in ONE-R and define the components of SDL.

- The object oriented data models such as ORIGON [76], Postgres [120] and O2 [52], provide the motivation to add behavioral aspects to ONE-R. Particularly, The O2 data model [17] and the language RELOOP [127], has strongly influenced the type expressions of ONE-R and its data language SDL.

Figure 3.2 shows a model of technical systems which is based on the ANSI-SPARC three leveled schema architecture. In the model, there is a database and some application systems accessing the database. According to the architecture, a data model must be able to support the three leveled schema architecture. We will redefine the concept of “scenario” [4] to support the definition of the schema levels.

Another problem is how to specify the behavioral properties of entities (objects). In the object-oriented languages, objects are often regarded as concurrently executing units [56]. A similar viewpoint is also adopted by OOD and OOA [24, 25]. However, since the data objects stored in a database are “passive”, there have been suggestions to treat “active” and “passive” objects differently [45, 133] because an active object can trigger an operation on other objects whereas a passive object can not. Therefore, a method defined on an entity type or a relationship type may be viewed as an operation which is triggered by other system components and will only work on the contents of an entity or relationship instance each time that it is invoked.

Finally, a good model should be both user friendly and formal in order to make it possible to check the consistency of designed schemas by mathematical methods. To reach this objective we also build a modeling structure, see
3.2.2 Basic components

ONE-R is used to specify the basic phenomena – entities and relationships among entities. We use the constructs entity class and relationship class. To express properties of the phenomena and possible operations on them, the constructs of data type and method are also provided.

An entity class is a set of entities (objects) that exist in the application domain. The information about entities is to be managed in the system activities that we are analyzing. In other words, an entity is a phenomenon in reality, whereas its image in our system is a value of an “abstract” type. An entity may be referred to by its identifier value.

An relationship class is a relation on two or more entity classes. Each relationship is also a phenomenon in reality. Its image in our system is a tuple of the identifier values of the corresponding entities.

Relationships to be defined among the classes are:

- is_a_sub_of is an “including” relationship between two entity classes. It means that all the members of the “sub” class are included in the “super” class. A subclass may also have subclasses, thus an is_a hierarchy can be formed. An entity class may be a subclass of more than one class, but it should actually belong to only one basic class. Furthermore, we
can define some constraints on the subclasses of an entity class: If an entity class $E$ has subclasses $S_1, \cdots, S_n$, these subclasses may make up $E$ ($S_1 \cup S_2 \cdots \cup S_n = E$), or be distinct such that for any $S_i, S_j, i \neq j, 1 \leq i \leq n, 1 \leq j \leq n, S_i \cap S_j = \phi$, or both (partition of $E$).

- *in_a_relationship* exists between an entity class and a relationship class. It shows that some members in an entity class may have relationships with some members of other entity classes or with those in the same entity class. There are four types to be specified for the entity class as a participant of the relationship class: *full_1, full_n, partial_1 and partial_n.* *full* means that any entity in the class must be involved in at least one occurrence of relationship whereas *partial* means that the involvement is not mandatory. $1$ and $n$ have the same meaning as that in the ER model.

In Figure 3.5 we draw several entity classes and relationship classes in the example of the *IFIP conference* [142] to show the concepts described above and their graphical notations of them. The IFIP conference example is worked through in detail in Chapter 6.

![Graphical representation of entity and relationship classes in IFIP conference](image)

**Figure 3.5: The Entity and Relationship Classes in ifip_conference**

In order to express properties of an entity or a relationship, *data* are used. A datum is an element of a *value set*. The value set, also called a *domain*, is represented in ONE-R by the construct *data type*. Several *primitive types*—integer, real, string (or string[n] where $n$ is the maximum length of a string) and boolean are pre-defined (Figure 3.6a). More complex data types can then be defined by the following constructors:

- *renaming* (Figure 3.6b): a new type $T$ is defined by renaming an existing type $T'$, i.e., any value of type $T$ is also of type $T'$;
• **compositing** (Figure 3.6c): a new type $T$ is a composite type with $n$ components of types $T_1, T_2, \cdots, T_n$. $\text{com.}1, \ldots, \text{com.}n$ are the names of the components. If a symbol "*" is attached to a component name, e.g., the $\text{com.}n$ in Figure 3.6c, then it is allowed that a composite value (of type $T$) has a null value for this component;

• **set** (Figure 3.6d): a new type $T$ is defined on the power set of the value set of type $T'$, so any value of type $T$ is a set of values of type $T'$. Two numbers may be attached to the definition for the minimum and maximum numbers of the elements of a value of type $T$. If the second number is actually a symbol "$n$", then no limitation on the maximum number is given. The default number pair is $(0, n)$;

• **union** (Figure 3.6e): a new type $T$ is the union of type $T_1, \cdots, T_n$ with $\text{label}_1, \cdots, \text{label}_n$, i.e., any value of type $T$ is either of type $T_1, \cdots$ or $T_n$.

![Diagram of data types in ONE-R](image)

**Figure 3.6: The data types in ONE-R**

In addition to the pre-defined primitive data types, the entity classes can also be regarded as **abstract data types**, and be used to construct new data types. However, this kind of data types has some parts that can not be printed, because the **identifier** of an entity is only a symbol reflecting the existence of
an entity within a database, so its value relies on the concrete DBMS and often can not be printed out as a meaningful value. We thus call a data type which is a primitive type or is constructed on the basis of primitive types pure data type to distinguish them from those that has at least one abstract data type as a part.

For an entity class or a relationship class, we can also define some methods that define possible operations on the members of the class. A method is a function which has one or several input values as its parameters and one output value as its result. When the method is invoked on an entity (or relationship) of the class, it will execute on the values of its parameters and the contents of the attributes of the entity (or relationship). We distinguish two kinds of methods according to whether a method updates the value of the attributes of the entity (or relationship) or not. We call the updating a side effect. In Figure 3.7 we give the graphical notations of the two kinds of methods. The implementation of methods may be given in programming languages such as C, PASCAL or PROLOG.

![Diagram](image)

Figure 3.7: Methods with their inputs and output

The properties of entities and relationships are described by attributes. An attribute is a function (total or partial) from an entity class or a relationship class to a data type. An attached symbol "**" to an attribute indicates that it is a partial function, i.e., for some entities (relationships), there may be a null value on this attribute. To define some operations on the entities or relationships, a set of methods can also be attached to the definition of classes. Like in most semantic and object-oriented models, the subentity classes author and referee inherit all the attributes and methods of their super class participant (see Figure 3.8).

In Figure 3.8 we give an example for the data model of the IFIP conference. It is assumed that only these entity classes and the relationship classes with the complex data types are dealt with in a data base.

The basic components of ONE-R provide users with the capability of specifying structural constraints, such as the manner of participation of entities in a relationship, the relations between sub-classes of a same entity class, etc. In addition to these, a user may want to express other specific constraints, e.g.,
Figure 3.8: The classes, data types and methods in the data model for the IFIP Conference example
the limitations on the values of the attributes of some entities. We enable the
users to do so by letting them write logical expressions in the (first order and
many sorted) language that is further described in the sequel.

3.2.3 The Formal Expressions of ONE-R

Now we represent the formal expressions of ONE-R which is used as the basis
for the definition of the ONE-R algebra, and which serves as a bridge be-
tween the user's view on an object world and its implementation by database
techniques. We call these formal expressions "internal forms" of ONE-R.

Why do we define the "internal form" of ONE-R? In Figure 3.9 the needs of
the internal form is depicted.

![Diagram of ONE-R model]

Figure 3.9: Mapping the reality into the computer world

Like most semantic data models, ONE-R can hardly be regarded as a pure data
model. Besides the data types, the concepts of entity and relationship reflect
our perception on the phenomena existing in the real world. However, within
the computer world, only data or symbols can be dealt with. Therefore, we must
map our perception on the real world to a pure data model within the computer
world. Only with the data model can we define data algebraic operations and use some logical theories to verify the correctness of our perception. In Figure 3.9 the entity types author and paper are reflected in the computer world through their ID types that are abstract types and may be implemented differently within different computer systems. Therefore, we have defined the relation-style representations of the entities and relationships, so as to provide a foundation for an algebra and a logical theory. 

The formal ONE-R model is built as a type structure. For this we have the following definitions:

**DEFINITION 1:** A Database Schema is a 6-tuple \((Tset, TDset, Dset, Aset, dom, C)\) such that

1. \(Tset\) is a set of type names;
2. \(TDset\) is a set of type expressions. The two sets \(Tset\) and \(TDset\) has a \(1:1\) mapping, thus for any \(T \in Tset\), there is a \(T_{des} \in TDset\), called the type expression of \(T\).
3. \(Dset\) is a set of value domains;
4. \(Aset\) is a set of names for attributes and labels;
5. \(dom : Tset \rightarrow Dset\) is a total function which maps a type name in \(Tset\) to a domain in \(Dset\);
6. \(C\) is a set of constraint expressions.

**DEFINITION 2:** A type \(T_{des}\) is a 5-tuple \((T, K, IDset, CONST, M)\) such that

1. \(T \in Tset\) is a type name;
2. \(K\) is an element of the word set \{"DATA", "ENTITY", "RELATIONSHIP", "CLASS", "DB"\};
3. \(IDset \subseteq Tset\) is a set of type names. These are Identity types used as some attributes of \(T\) to uniquely determine a value of type \(T\) (for an entity type the set contains only one Identity type);
4. \(CONST\) is a type constructor;
5. \(M\) is a set of method expressions.

**DEFINITION 3:** A type constructor \(CONST\) can be one of the following expressions:

1. \(\emptyset\);
2. \((T')\) (we call it renaming constructor);
3. \((A_1 : T_1, \cdots, A_n : T_n)\), \(A_i \in Aset, T_i = T'_i\) or \(T'_i^\ast, T'_i \in Tset, i = 1 \cdots n\) (we call it tuple constructor, the symbol "\(\ast\)" indicates that the attribute
may have a null value);
4. \( \{T'\}(m_1 : m_2) \), \( m_1 \) is a number while \( m_2 \) may be a number or a letter "n" (we call it set constructor);
5. \( \bigcup(A_1 : T_1 \cdots A_n : T_n), A_i \in Aset, T_i \in Tset, i = 1 \cdots n \) (we call it type union constructor).

**DEFINITION 4:** A method expression is a 3-tuple \((m, k, sig)\) such that

1. \( m \) is the name of the method.
2. \( k \) is an element of the word set \{"no\_side\_effect", "has\_side\_effect"\};
3. \( sig \) is a formula \( T_0 \times T_1 \cdots \times T_n \rightarrow T_s \), where \( T_i, T_s \in Tset, i = 0, \cdots n \).

\( C \) contains constraints of two categories:

1. **structural constraints** are expressions that show some required properties of the types in the database schema:
   - \( IS.A(T', T) \) (showing that \( \text{dom}(\text{Id.of.T'}) \subseteq \text{dom}(\text{Id.of.T}) \);
   - \( \text{DISTINCT}(T_1, \cdots T_n) \) (showing that \( \forall i \forall j(i, j = 1, \cdots, n, i \neq j), \text{dom}(\text{Id.of.T}_i) \cap \text{dom}(\text{Id.of.T}_j) = \emptyset \);
   - \( \text{MAKING.UP}(T_1, \cdots, T_n, T) \) (showing that \( \text{dom}(\text{Id.of.T}) = \text{dom}(\text{Id.of.T}_1) \cup \cdots \cup \text{dom}(\text{Id.of.T}_n) \));
   - \( \text{TYPE.OF.RELATIONSHIP}(R, P_1, \cdots, P_n) \) (showing that \( R \) is a \( n \)-ary relationship type and \( P_i(i = 1, \cdots, n) \) is the manner with that the \( i \)th participant of \( R \) participates the relationship type. \( P_i \) is either full\(_1\), partial\(_1\), full\(_n\) or partial\(_n\)).

2. **semantic constraints** are logical formulae to express the constraints on the values stored in a database. The logical language will be formally defined in the next chapter, but is also shown in Section 3.2.4 for the data algebra of ONE-R.

It is worth mentioning that the *inheritance* between an entity type \( T \) and a sub-type \( T' \) should be reflected in the type expressions. First, we let the type expressions of \( T' \) contain all attributes of its parent type. This is not enough. An entity sub-type must also inherit the relationships that its parent type is involved in. Therefore, for any relationship type \( R \) whose participant types include \( T \), the type expressions for a new relationship type \( R' \), as well as the structural constant expression \( \text{TYPE.OF.RELATIONSHIP}(R', \cdots) \), will also be created and put into the database schema.

To explain the meaning of the DB schema, we will illustrate the process of building the *type structure* (or the *type hierarchy*), for the IFIP example. The example was made in the early 1980s by the IFIP 8.1 working group as a benchmark to evaluate and compare different modeling approaches [142]. In this
chapter we use a simplified specification for the data & entity & relationship
types in the example (see Figure 3.8). A more complete description of the
example is given in Chapter 6, in which all aspects of information management
in the IFIP conference are illustrated.

First we define the primitive data types. We say that type $T$ is a primitive
data type if the type expression for $T$ is

$$(T, "DATA", \emptyset, \emptyset)$$

It means $T$ has an empty identity type set as well as an empty type constructor.
Besides, no method is defined on $T$.

The primitive data types in our example are:

- The pre-defined data types $INTEGER, REAL, STRING, BOOLEAN$,
- For each entity class, there is an identity type which is a data type and
  is to be implemented in a concrete system. In the example, the identity
types are $Id\_participant, Id\_of\_author, Id\_of\_referee, Id\_of\_paper$. A value
  of such a type is the image of an entity in the system, thus the domains
  of the identity types are decided by the supporting DBMSs.

All other data types will have a type expression of the form

$$(T, "DATA", \emptyset, CONS_T, \emptyset)$$

where $CONS_T \neq \emptyset$. Therefore, we need only list their type constructors:

- $CONS_{name1} = (first\_name : STRING,
  middle\_name : STRING, last\_name : STRING)$;
- $CONS_{name2} = (first\_name : STRING,
  last\_name : STRING)$;
- $CONS_{name} = \bigcup (name1 : name1, name2 : name2)$;
- $CONS_{date} = (year : INTEGER,
  month : INTEGER, day : INTEGER)$;
- $CONS_{languages} = \{STRING\}(1 : n)$;
- $CONS_{authors} = \{Id\_of\_author\}(0 : n)$;
- $CONS_{keywords} = \{STRING\}(0 : n)$;
- $CONS_{front\_page} = (authors : authors,
  keywords : keywords, abstract* : STRING)$;
3.2. The Extended Entity-Relationship Model (ONE-R)

- \( CONS_{\text{publication}} = (\text{kind} : \text{STRING}, \text{title} : \text{STRING}) \);

- \( CONS_{\text{publications}} = \{\text{publication}\}(0 : n) \);

- \( CONS_{\text{study}} = (\text{direction} : \text{STRING}, \text{period} : \text{STRING}, \text{publications} : \text{publications}) \);

- \( CONS_{\text{studies}} = \{\text{study}\}(0 : n) \);

Their domains are decided by the following rules:

1. If \( CONS_T = (T') \), then \( \text{dom}(T) = \text{dom}(T') \);
2. If \( CONS_T = (A_1 : T_1, \ldots, A_n : T_n) \), then
   \[ \text{dom}(T) = \text{dom}(T_1) \times \cdots \times \text{dom}(T_n) \];
3. If \( CONS_T = \{T\}(m_1 : m_2) \), then \( \text{dom}(T) \equiv \{s \mid s \in P(\text{dom}(T')) \land (m_1 \leq \text{card}(s) \leq m_2) \text{ or } (m_1 \leq \text{card}(s) \text{ when } m_2 \text{ is } "n" ) \} \);
4. If \( CONS_T = \bigcup (A_1 : T_1, \ldots, A_n : T_n) \), then \( \text{dom}(T) \equiv \text{dom}(T_1) \cup \cdots \cup \text{dom}(T_n) \).

After all the data types in the example have been defined, we can define the entity types and relationship types:

- the type expression of \text{participant} is
  \( (\text{participant}, \text{"ENTITY"}, \{\text{Id.of participant}\}, CONS_{\text{participant}}, \{m_1, m_2\}) \)
  where
  \( CONS_{\text{participant}} = (\text{id} : \text{Id.of participant}, \text{name} : \text{name}, \text{birth.date} : \text{date}, \text{address} : \text{STRING}) \) and
  \( m_1 = (\text{get.age}, \text{"no.side.effect"}, \text{participant} \times \text{date} \rightarrow \text{INTEGER}) \)
  \( m_2 = (\text{change.birth.date}, \text{"has.side.effect"}, \text{participant} \times \text{date} \rightarrow \text{date}) \).

- the type expression of \text{author} is
  \( (\text{author}, \text{"ENTITY"}, \{\text{Id.of author}\}, CONS_{\text{author}}, \{m_1, m_2\}) \)
  where
  \( CONS_{\text{author}} = (\text{id} : \text{Id.of author}, \text{name} : \text{name}, \text{birth.date} : \text{date}, \text{address} : \text{STRING}, \text{language} : \text{languages}, \text{native.language} : \text{STRING}) \).

- the type expression of \text{referee} is
  \( (\text{referee}, \text{"ENTITY"}, \{\text{Id.of referee}\}, CONS_{\text{referee}}, \{m_1, m_2\}) \)
  where
  \( CONS_{\text{referee}} = (\text{id} : \text{Id.of referee}, \text{name} : \text{name}, \text{birth.date} : \text{date}, \text{address} : \text{STRING}, \text{study.experiences} : \text{studies}) \).
• the type expression of paper is
  \((\text{paper}, \text{"ENTITY"}, \{\text{Id}_\text{of\_paper}\}, \text{CONS}_\text{paper}, \{\})\)
  where
  \(\text{CONS}_\text{paper} = (id : \text{Id}_\text{of\_paper}, \text{title} : \text{STRING},\)
  \(\text{front\_page} : \text{front\_page}, \text{language} : \text{STRING}).\)

• the type expression of write is
  \((\text{write}, \text{"RELATIONSHIP"}, \{\text{Id}_\text{of\_author}, \text{Id}_\text{of\_paper}\},\)
  \(\text{CONS}_\text{write}, \{\})\)
  where
  \(\text{CONS}_\text{write} = (id1 : \text{Id}_\text{of\_author}, id2 : \text{Id}_\text{of\_paper}).\)

• the type expression of review is
  \((\text{review}, \text{"RELATIONSHIP"}, \{\text{Id}_\text{of\_referee}, \text{Id}_\text{of\_paper}\},\)
  \(\text{CONS}_\text{review}, \{\})\)
  where
  \(\text{CONS}_\text{review} = (id1 : \text{Id}_\text{of\_referee}, id2 : \text{Id}_\text{of\_paper},\)
  \(\text{mark} : \text{INTEGER}).\)

Since all the entity types and relationship types are defined by the tuple constructor, their values are a tuple consisting of the values of the underlying types. For instance,

\[
\text{dom(participant)} = \text{dom(}\text{Id}_\text{of\_participant}) \times \text{dom(name)} \times \text{dom(date)} \times \text{dom(STRING)}.
\]

Now we will define the class types for the entity and relationship types. In ONE-R we define a class type for each entity or relationship type. The type expression for such a class type \(T\) is

\((T, \text{"CLASS"}, \{\}, \text{CONS}_T, M)\)

where \(M\) is a pre-defined operations such as insert, update, delete etc. The type constructors for the entity types and relationship types in our example are

• \(\text{CONS}_\text{Class\_of\_participant} = \{\text{participant}\}(0,n)\)

• \(\text{CONS}_\text{Class\_of\_author} = \{\text{author}\}(0,n)\)

• \(\text{CONS}_\text{Class\_of\_referee} = \{\text{referee}\}(0,n)\)

• \(\text{CONS}_\text{Class\_of\_paper} = \{\text{participant}\}(0,n)\)

• \(\text{CONS}_\text{Class\_of\_write} = \{\text{write}\}(0,n)\)

• \(\text{CONS}_\text{Class\_of\_review} = \{\text{review}\}(0,n)\)
3.2. The Extended Entity-Relationship Model (ONE-R)

Obviously, any value of a class type is a set of values of an entity type or a relationship type, i.e., a tuple set.

Finally, we define the DB type:

- The type expression of DB is
  $$(DB,"DB",\{\}, CONS_{DB}, MA)$$
  where $MA$ is the algebraic operator set that we will introduce in the next section, and

- $CONS_{DB} = (\text{class}_1 : Class\_of\_participant^*, \text{class}_2 : Class\_of\_author^*,$
  $\text{class}_3 : Class\_of\_referee^*, \text{class}_4 : Class\_of\_paper^*,$
  $\text{class}_5 : Class\_of\_write^*, \text{class}_6 : Class\_of\_review^*)$$

According to the type constructor,
$$dom(DB) \equiv dom(CLASS_1) \times \cdots \times dom(CLASS_c)$$

Thus a value of type $DB$ is a tuple whose attributes are sets of entity or relationship tuples. We also call such a tuple the state of a database.

The constraints enforced on any DB state is defined as the constraint expressions in $C$:

1. $IS\_A(author, participant);$
2. $IS\_A(referee, participant);$
3. $MAKING\_UP(author, referee, participant);$
4. $TYPE\_OF\_RELATIONSHIP(write, full_n, full_n);$
5. $TYPE\_OF\_RELATIONSHIP(review, partial_n, full_n);$

In addition to the formal expressions, we also provide a set of graphical notations for the internal forms of ONE-R. In Figure 3.10 we show the shapes of ONE-R type constructors, then in Figure 3.11, Figure 3.12 and Figure 3.13 the IFIP example is shown.

The process illustrated above builds a four-level type hierarchy, as that shown in Figure 3.14. The structure is quite similar to that of a relational database: The data types correspond to the attributes in RDBs, the entity and relationship types correspond to the relation types (tuple types), the classes correspond to the tables, and the DB type correspond to the collection of tables, i.e., the database contents. In this sense, we call the internal form of ONE-R the reshaped relational model consisting of a type structure. Whilst the designers describe their object world by the abstract concepts of entity class and relationship class, the type hierarchy is a pure data model which expresses all the abstract concepts in terms of data types. We further illustrate
Figure 3.10: The ONE-R type constructors

Figure 3.11: The data types in the IFIP example
3.2. The Extended Entity-Relationship Model (ONE-R)

<table>
<thead>
<tr>
<th>participant</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>ID_OF_PARTICIPANT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>author</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>ID_OF_AUTHOR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>referee</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>ID_OFREFEREE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>ID_OF_PAPER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID1</td>
</tr>
<tr>
<td>ID_OF_AUTHOR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>review</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID1</td>
</tr>
<tr>
<td>ID_OFREFEREE</td>
</tr>
</tbody>
</table>

Figure 3.12: The internal forms of the entity and relationship types in the IFIP example
the relations between the basic ONE-R constructs given in Section 3.2.2 and the type structure in Section 3.2.3 as follows:

- all the data types along with their type constructors are kept directly in the type structure, except that the *compositing* constructor is replaced by a *tuple* constructor, and the name *component* is now also called *attribute*. Obviously the meaning of the constructor is not changed.

- For every entity class \( E \), an identity data type \( ID.\_E \) is defined so that a member of the class can be represented by a value of type \( ID.\_E \). Also, an entity type for the class is defined (called \( E \) too) as a tuple type which consists of the identity attribute on the type \( ID.\_E \) and all the attributes of the entity class, thus a value of the entity type is a tuple composed of the values on the identity attribute and all other attributes. In addition, a class type \( CLASS.\_E \) is defined, which has only one value at any time: the value is a set of tuples of the entity type. For convenience, we also use \( \{E\} \) to denote the value of type \( CLASS.\_E \). Besides, if any entity class has taken part in a type construction as an abstract data type, e.g., type *author* for type *authors* in Figure 3.8, then in the type hierarchy it is replaced by its identity type.

- For any relationship class \( R \), a relationship type is also defined (called \( R \) too) as a tuple type which consists of the identity types for the relevant entity classes and the attributes of the relationship class itself, so that a value of type \( R \) is a tuple composed of several identity values and other data values. Also, a class type \( CLASS.\_R \) is defined, whose only value at any time is a set of tuples of type \( R \). For convenience, we also use \( \{R\} \) to denote the value of type \( CLASS.\_R \).

- A DB is defined as the collection of all the classes, so any any time there is only one value of the type: the database itself.
3.2. The Extended Entity-Relationship Model (ONE-R)

Because ONE-R is also intended to be an object oriented model, the behavioral aspects should be dealt with. In the formal type hierarchy of ONE-R, we do not attach any method to data types while all the user-defined methods for entity classes and relationship classes are attached to the entity and relationship types. On the class type level, we attach the data manipulation methods: insert, delete and update, etc. On the DB type level, we attach a set of ONE-R algebraic operators that we will introduce in the next section.

Finally, the type structure expressed in this section is still a conceptual model and not necessarily in the implementation form. In a practical implementation, the common attributes of the entity classes are often put on the relations for super classes while the special attributes of the sub-entities are put on the relations for them. In [69], such an implementation is described. In [84], Kung suggested to use a view facility to realize the super-subclass hierarchies.

3.2.4 The Algebra of ONE-R

Now we can define the ONE-R algebra on the value sets of the types defined by the formal type structure that we depicted in the last section. Prior to a detailed description, we first give a brief introduction to the characteristics of the ONE-R algebra:

- The operator set is \( \{ \cup, -, \times, \sigma, \pi, \rho, \nu, \mu, \alpha, \beta \} \);
Chapter 3. COMIS – an integrated system modeling approach

- To any operator, its operands are value sets whose elements are of either entity types, relationship types, or data types while its result is a value set whose elements are of a data type. Since any operation is invoked with the values of entity class types and/or or relationship class types, i.e., the tuple sets, the results of the calculation must also be a tuple sets, so we also call the ONE-R algebra a reshaped algebra.

Further, we explain the operators as follows:

1. union "U":

   \[ C = C_1 \cup C_2 = \{ t \mid t \in C_1 \lor t \in C_2 \} \]

2. difference "−":

   \[ C = C_1 - C_2 = \{ t \mid t \in C_1 \land t \notin C_2 \} \]

These two operators generate results that have the same type as their operands.

3. product "×":

   \[ C = C_1 \times C_2 = \{ t \mid t = (A_1 : v_1, \ldots A_n : v_n, A'_1 : v'_1, \ldots A'_m : v'_m) \land \exists t_1 \exists t_2 (t_1 = (A_1 : v_1, \ldots A_n : v_n) \land t_2 = (A'_1 : v'_1, \ldots A'_m : v'_m)) \} \]

When the type of the elements in \( C_1 \) and \( C_2 \) are tuple types with attributes \( A_1, \ldots, A_n \) and \( B_1, \ldots, B_m \) respectively, then the type of the elements in the result set is also a tuple type (Figure 3.15).

<table>
<thead>
<tr>
<th>MC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 : Ta1</td>
</tr>
</tbody>
</table>

Figure 3.15: The result type of the product operator

4. selection “σ”:

   \[ C = \sigma_{F(x)}(C_1) = \{ t \mid t \in C_1 \land ((F(t) \text{ is true})) \} \]

Like in other models, we use a wff \( F(x) \), called condition formula, to express the selection condition, where \( x \) is a free variable whose domain is the value set to which the operator is applied. \( F(x) \) is written in a normal logical form:

1. the atomic parts of \( F(x) \) (predicates) are of the form "term1 cop term2" where
3.2. The Extended Entity-Relationship Model (ONE-R) 75

- $\textit{cop} \in \{<, \leq, >, \geq, =, \neq, \subseteq, \supseteq, \subseteq, \supseteq, \in, \notin\}$

- A term may be a constant value in primitive, tuple, set or a nested form, an attribute sequence $x.A_1 \cdots A_k$ (the prefix $x.$ can be omitted), a variable $y (y \neq x)$, the cardinal of a set value $\text{card}(\cdots A_i)$, or a method calling $m(v_1, \cdots, v_m)$ where $m$ is the name of the method and $v_1, \cdots, v_m$ are the parameters of the method. $m$ must not have side-effects.

2. In addition to the logical symbols $\sim, \lor, \land$ and $\rightarrow$, the quantifiers $\exists$ and $\forall$ are used to quantify the variables. Any variable except $x$ must be quantified in $F(x)$ so that when $x$ is substituted by a tuple value $t$, $F(t)$ is a closed formula. Besides, the expression $A_i \cdots A_k \{y\}$ can be used to substitute the expression $y \in A_i \cdots A_k$.

As an example,

\[
\sigma_{\text{language} = \text{English}} \land \exists k(x.\text{front\_page\_keywords}(k) \land k = \text{"datamodel"}) \{\{\text{referee}\}\}
\]

is a selection operation which finds all the papers containing the keyword “data model” in its front page. The selection operator also generate the same result type with its operand.

5. Projection “$\pi$”:

\[
C = \pi_E(C_1) = \{t \mid \exists t' t' \in T_1 \land p(t') = t\}
\]

where $E$ is a projection expression and $p$ is a function to map any tuple $t'$ in $C_1$ to a tuple $t$ which contains the values on the parts indicated by $E$.

The projection operator is used to get some values of a tuple set on a subset of its attributes. In the relational data model, the attributes are expressed in a project list ($A_1 \Rightarrow B_1, \cdots A_k \Rightarrow B_k$). However, Schek and Scholl has found that for the nested data types, it is not unsatisfactory to only take an attribute as a whole or nothing by a projection on a tuple type, so the compound projection operator is defined [129]. Parent and Spaccapietra has also suggested that the projected attributes be possibly at any level in the complex data structure [114, 115]. Therefore, we must provide projection expressions in the possible nested forms. Follows is a BNF-like grammar for projection expressions:

P1. project_expression ::= (project_list)
P2. project_list ::= project_term1, \cdots, project_term_n
P3. project_term ::= project_item
P4. project_term ::= project_head (project_list)
P5. project_term ::= project_head project_set_term
P6. project_term ::= project_head \cup (project_list)
P7. project_set_term ::= \{ (project_list) \}
P8. project_set_term ::= \{ \cup (project_list) \}
P9. project_set_term ::= \{ project_set_term \}
P10. \texttt{project\_head ::= attribute\_name_1 ⇒ attribute\_name_2}

P11. \texttt{project\_item ::= attribute\_name_1 ⇒ attribute\_name_2}

P12. \texttt{project\_item ::= method\_name(\texttt{parameter\_list}) ⇒ attribute\_name}

As an example, the operation

\[
\{\texttt{referee1}\} = \pi_{\{\texttt{name}⇒\texttt{referee\_name}, \texttt{study\_experiences}⇒\texttt{researches}\}}
\{
\{\texttt{direction}⇒\texttt{direction}, \texttt{publications}⇒\texttt{papers} \{\{\texttt{title}⇒\texttt{paper}, \texttt{kind}⇒\texttt{published\_in}\}\}\}
\}\{\texttt{referee}\}
\]

is a legal projection. The type \texttt{referee1} of the members in the result set is shown in Figure 3.16.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\texttt{referee\_name} & \texttt{researches} \\
: name & : \texttt{studies'} \\
\hline
\texttt{---------} & \texttt{study'} \\
 & \texttt{papers} : \texttt{publications'} \\
 & : \texttt{STRING} \\
 & \{ \texttt{publication'} \\
 & \{ \texttt{paper} : \texttt{STRING} \\
 & \texttt{published\_in} : \texttt{STRING} \\
\hline
\end{tabular}
\caption{Figure 3.16: The result type of a project operation}
\end{figure}

\section{reduction \textdprime{\textit{\rho}}:}

\[
C = \rho_R(C_1) = \{ t \mid \exists t'(t' ∈ C_1 ∧ (\text{the value of } t \text{ is the left part of } t'
\text{ after the the unwanted elements in a set value are removed})\}
\]

The reduction operator maps a tuple to a new tuple where the unwanted elements within a set value are removed. A reduction expression \(R\), in the form \((x : F)\), is used. \(x\) is a free variable in the formula \(F\) and its domain is the elements of a set type value that is a part of a tuple. The formula \(F\) will specify the wanted properties of the variable. When we use a value to substitute \(x\) and \(F\) is not true with the substitution, then the value or part of the value is removed.

It is possible that there are more than one set type in the type hierarchy. In this case, it is required that all the variables for the set types above the type of \(x\) are \textit{free}, and all the variables below the type of \(x\) are \textit{quantified}. For instance,
to remove the publications of the referees that are of the “report” kind, we can have following reduction operation:

\[ \rho(p \text{. study.experiences}(s) \land s \text{. publications}(p) \land p \text{. kind} \neq \text{"report"})\{(\text{referee})\} \]

The type of the result the operator is the same as that of the the operand.

7. **nest** "\(\nu\)". This operator nests several attributes in a part of the data type structure (of a complex type) into one attribute:

\[ C = \nu_{\text{nest item}}(C_1) = \{ t | \exists t' t' \in C_1 \land t' = (A_1 : v_1, \cdots, *A_i : v_i, \cdots, A_{i+k} : v_{i+k}, \cdots, A_n : v_n) \land t = (A_1 : v_1, \cdots, A : v_A, \cdots, A_n : v_n) \land v_A = (A_i : v_i, \cdots, A_{i+k} : v_{i+k}) \} \]

The pair of "*" symbols is the possible embedded structure where \(A_i, \cdots, A_{i+k}\) is located. This means that the nest operation can be applied to any level of the type structure of the operand. For a more precise description we give the grammar of the nest.item:

**P1.** nest.item ::= ((A_i, \cdots, A_{i+k}) \Rightarrow A)

**P2.** nest.item ::= A_i.nest.item

**P3.** nest.item ::= {nest.item}

For instance, The operation

\[ (\{\text{referee}\}) = \nu_{(\text{name,birth.date,address})\Rightarrow\text{general.information}}(\{\text{referee}\}) \]

will generate the result of the type in Figure 3.17.

<table>
<thead>
<tr>
<th>referee</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID : ID_OFREFEREE</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>: name</td>
</tr>
</tbody>
</table>

**Figure 3.17:** The result type of a nest operation

8. **unnest** "\(\mu\)". This operator unnests an attribute and replaces it by its attributes within the type structure of a complex type:

\[ C = \mu_{\text{unnest.item}}(C_1) = \{ t | \exists t' t' \in C_1 \land t' = (A_1 : v_1, \cdots, *A : v_A, \cdots, A_{i+k} : v_{i+k}) \land t = (A_1 : v_1, \cdots, *A : v_A, \cdots, A_{i+k} : v_{i+k}) \} \]

and the grammar for the unnest.item is:
P1. unnest_item ::= A
P2. unnest_item ::= Ai.unnest_item
P3. unnest_item ::= {unnest_item}

The operation

\((\text{\{referee\}}) = \mu_{\text{general_information}}(\text{\{referee\}})\)

will generate a result with the type as the referee in Figure 3.13.

9. pack "\(\alpha\)": The \(\alpha\)-operator groups the values of a tuple type (in the type structure of a complex type) in such a way that the values in a group have the same values for all attributes except the specified one. The values of the specified attribute in this group will be "packed" into a set value and the other values will be replaced by one value (because they are same). The packed attribute itself may also be renamed. The operation may be applied to any level of the complex type of the operand. If the packing happens at the highest level, then the number of tuples will be reduced. The operator is defined as

\[
C = \alpha_{\text{pack.item}} C_1 = \{ t \mid t = (A_1 : v_1, \ldots, *B_i : v_i, \ldots, A_n : v_n) \wedge \\
\forall t' \in C_1(t' = (A_1 : v'_1, \ldots, *A_i : v'_i, \ldots, A_n : v'_n) \wedge \\
(v'_i = v_i \land \cdots \land v'_n = v_n \land \cdots) \land v'_i \in v_i) \}
\]

and the grammar for the pack_item is:

P1. pack_item ::= (A_i \Rightarrow B_i)
P2. pack_item ::= Ai.pack_item
P3. pack_item ::= {pack_item}

A new type is also obtained through the operation. An example is given in Figure 3.18 which also shows the result of a composite operation.

10. unpack "\(\beta\)". This operator "unpacks" the values in a set value on an attribute of a tuple type as a part in the type structure of a complex type. For each of the values, there is a result tuple which combines the value with the values of the tuple on other attributes. Like the pack operation, it can also be applied to any level of the type structure. If it is applied to the highest level of the complex type, then the number of tuples of the type may be increased. The operator is defined as

\[
C = \beta_{\text{unpack.item}} C_1 = \{ t \mid t = (A_1 : v_1, \ldots, *A_i : v_i, \ldots, A_n : v_n) \wedge \\
\exists t' \in C_1(t' = (A_1 : v'_1, \ldots, *B_i : v'_i, \ldots, A_n : v'_n) \wedge \\
(v'_i = v_i \land \cdots \land v'_n = v_n \land \cdots) \wedge v'_i \in v_i) \}
\]

and the grammar for the unpack_item is:

P1. unpack_item ::= (B_i \Rightarrow A_i)
P2. unpack_item ::= Ai.unpack_item
3.2. The Extended Entity-Relationship Model (ONE-R) 79

P3. \texttt{unpack\_item ::= \{unpack\_item\}}

The operation is the inverse operation of \textit{pact}, thus the type of the result is also changed.

We have described the ten operators that make up the ONE-R algebra. There are some operations that can be expressed in terms of the basic operators and we call them \textit{macro operators}. We list some most frequently used macro operators below:

11. \textbf{join “⊗”:}

\[ C_1 \bowtie C_2 = \sigma_{A \theta B}(C_1 \times C_2) \]

where \(A\) and \(B\) are attributes of the elements in \(C_1\) and \(C_2\) respectively, and \(\theta\) is a comparison symbol listed above for the selection operator \(\sigma\).

12. \textbf{natural join “*”:} If the elements in \(C_1\) and \(C_2\) have common attributes \(A_i\) and \(B_j\) defined on a same domain, then

\[ C_1 \ast C_2 = \pi_{(A_1, \ldots, A_n, B_1, \ldots, B_{j-1}, B_{j+1}, \ldots, B_m)}(\sigma_{A_i = B_j}(C_1 \times C_2)) \]

where \(A_k(0 \leq k \leq n)\) and \(B_l(0 \leq l \leq n)\) are attributes of the elements in \(C_1\) and \(C_2\) respectively.

13. \textbf{relationship join “*\(_R\)”}: When \(E_1, E_2, \ldots, E_n\) are entity types and \(R\) is a relationship type defined on them, then the relationship join

\[ \{E_1\} *_R \{\{E_2\}, \ldots, \{E_n\}\} = \{E_1\} * ((\cdots ((\{R\} \ast \{E_2\}) \ast \cdots) \ast \{E_n\}) \]

14. \textbf{packed relationship join “⊙\(_R\)”}: When \(E_1, E_2, \ldots, E_n\) are entity types and \(R\) is a relationship type defined on them, \(A_{11}, \ldots, A_{1m}\) are the attributes of \(E_1\), \(A_{21}, \ldots, A_{2m}\) are the attributes of \(E_2\), \(\ldots, A_{n1}, \ldots, A_{nn}\) are the attributes of \(E_1\) and \(A_{R1}, \ldots, A_{Rm}\) are the attributes of \(R\), then the packed relationship join on \(E_1, \ldots, E_n\) and \(R\) is

\[ \{E_1\} \odot_{R: RA} \{\{E_2\}, \ldots, \{E_n\}\} = \alpha_{(RA \Rightarrow RA)}(\nu_{(A_{21}, \ldots, A_{2m}, \ldots, A_{n1}, \ldots, A_{nn}, A_{R1}, \ldots, A_{Rm}) \Rightarrow RA}((\{E_1\} *_R \{\{E_2\}, \ldots, \{E_n\}\}))) \]

This operation is similar to the “relation join” defined by Parent and Spaccapietra \[114, 115\]. It pack the entity values in relationships with the same instance of another entity type. For instance, if we want to group all papers refereed by the any referee, we can have following operation:

\[ \{\textit{referee\_and\_papers}\} = \{\textit{referee}\} \odot_{\textit{review:refereed\_papers}} \{\{\textit{paper}\}\} \]
The result type is shown in Figure 3.18. This new type also shows the effect of the *pack* operation.

![Diagram](image)

Figure 3.18: The result type of a packed relationship join operation

### 3.2.5 The Semantic Data Language

We have defined the Semantic Data Language (SDL) for ONE-R. SDL is an SQL style language which supports data definition, manipulation and query in terms of our modeling concepts. Meanwhile, the query statements have their algebraic semantics defined by the operators introduced in the last section. We describe SDL in the sequel.

#### Data Definition

According to the ANSI-SPARC report [31], the description of a database consists of three levels of schemas: physical schema, conceptual schema and external schema. To give the corresponding structure in PM, Sälvberg initially introduced the concept "scenario" [4] for describing the user's external perspectives on a database: each scenario contains some entity classes linked by relationship classes, as to describing a piece of the universe of discourse. In SDL, we use *build_scenario* statements for the purpose. The following example defines the *IFIP-scenario*(Example 1):

**Example 1:**

```sql
build scenario ifip

data type name1 = (first_name: STRING, 
middle_name: string, last_name: STRING);
data type name2 = 
  (first_name: STRING, last_name: STRING);
data type name = 
  U(name1: first_name, name2: last_name);
```
data type date = (year: INTEGER,  
    month: INTEGER, day: INTEGER);

entity class participant =  
{  
    attributes:  
        name: name;  
        birth_date: private date;  
        address: STRING;  
    methods:  
        get_age;  
        change_birth_date;  
};

data type languages = {STRING};

entity class author =  
{  
    is_a_subclass_of: participant;  
    attributes:  
        speak: languages;  
        native_language: STRING;  
};

data type publication =  
    (kind: STRING, title: STRING);  
data type publications = {publication};
data type study = (direction: STRING,  
    period: STRING, publications: publications);  
data type studies = {study};

entity class referee =  
{  
    is_a_subclass_of: participant;  
    attributes:  
        study_experiences: studies;  
};

data type keywords = {STRING};
data type authors = {author};
data type front_page = (authors: authors,  
    keywords: keywords, abstract*: STRING);  
/* there may be a null value for "abstract"*/

entity class paper =  
{  
    attributes:
title: STRING;
front_page: front_page;
language: STRING;
};

relationship class review =
{
    on_entity_classes:
        referee, paper: (n:n) (p:f);
};

relationship class write =
{
    on_entity_classes:
        author, paper: (n:n) (f:f);
    attributes:
        mark: INTEGER
};

method get_age(Input_date): INTEGER;
    Input_date: date;
    {
        return(Input_date.year - birth_date.year);
    }

method change_birth_date(Input_date): INTEGER;
    Input_date: date;
    {
        birth_date := Input_date;
        return( 0 );
        /* only shows that the operation succeeded*/
    }

constraint c1:
    author, referee
    make up participant;

end

In the example, we see:

- We can define data types, entity classes, relationship classes, methods, and the constraints on the conceptual components.
- We can specify that an attribute of an entity or relationship class is **private** so that no direct access on it is allowed and only some **methods** can access it, otherwise the attribute is **public** and can be accessed freely.
The user-defined methods can only be attached to the members of entity or relationship classes.

We do not need to define the identity attribute for every entity class, since the identity attribute will be defined automatically by the support system whenever the entity class is created.

An entity class may also be regarded as a data type. In this case, its identity attribute will be actually used.

In SDL, a database schema will in general reflect the union of several scenarios. Each scenario describes a piece of the reality world. Besides the statement given in Example 1, we can also use the operations of merge scenario, modify scenario and remove scenario to modify a database schema.

In addition to the definition of scenarios, SDL also supports the definition of external schemas. The build external scenario statement can be used for this purpose (Example 2):

**Example 2:**

```sql
build external scenario ifip.at.Spain
  referencing ifip

accessible classes:
  participant, author, referee, paper,
  write, review;

aliases:
  person = participant;

derived classes:
  nice_participant =
    select *
    from participant
    where speak contains
      { "English", "Spanish"};
end
```

As we see from the example, an external scenario may reference other scenarios and have some accessible classes from them. In addition, aliases and derived classes can also be defined.

By using the DDL part of SDL, we can build a three-level schema hierarchy (see Figure 3.19) which fits to the information system model in Figure 3.2.
Data Query

A query statement in SDL works on the entity classes and relationship classes, but only returns a complex tuple data set as the result of the query. The statement has the following format:

```
select sel_expr_list
from from_list
[ where boolean ]
[ group by group_items ]
[ order by order_items ]
```

The SDL query statement is very similar to that of SQL. However, since ONE-R is a semantic and object-oriented language, many semantic concepts have been integrated in the various parts of the query statement. We will show them in the following examples.

Example 3:

```
select name,
   age = get_age(1992, 7, 1),
   study_experiences{z}
   ( period, publication{y}
     ( title where y.kind = "journal paper")
   where x.direction = "computer"
) 
```
from r in referee;

In this query we collect the name and age of referees with the their work period and the papers published in journals in the computer field. In the select part, we see:

- methods without side-effects can be invoked;
- on a set value attribute a condition can be specified to reduce the unwanted values from the set. The reduction can be done in multi levels.

Example 4:

```sql
select name, writes(title, language) from a in author, p in paper where a relates p through write group by a;
```

The query gets the name of the authors with the titles and languages of the papers they wrote. In the where part, the relationship write is used as the explicit condition, and the results are grouped by the authors, i.e., a author’s name is chosen together with the papers he wrote.

Example 5 shows the renaming on attributes and the changing of their order:

Example 5:

```sql
select author_name=
  long_name = name1
  (family_name = last_name,
   given_name1 = middle_name,
   given_name2 = first_name,
  ),
  short_name = name2
  (family_name = last_name,
   given_name1 = first_name,
   given_name2 = ""
  )
from author;
```

A query statement has its algebraic semantics defined by the operations introduced in the previous section. For example, the query in Example 4 can be
expressed by following algebraic operations:
\[
\pi_{\text{name}, \text{address}}(\text{participant}) \cup \{ \text{author} \} \circ \text{write}(\{ \text{paper} \})
\]

Data Manipulation

The manipulation part of SDL includes the statements of \textit{insert}, \textit{delete}, \textit{update}, \textit{add\_relationship} and \textit{remove\_relationship}. The first three statements are similar to those in SQL, but we also add more clauses into them for the semantic constraints in ONE-R. Data manipulation in SDL is expanded through two examples.

Example 6:

\textbf{insert into} participant\( (\text{name, address}) \)
\textbf{values} \( \{\text{name1} (\text{first\_name: } \text{"Odd"}, \text{middle\_name: } \text{"Ivar"}, \text{last\_name: } \text{"Lindland"}, \text{"IDT/NTH, 7034 Trondheim, Norway"}) \}
\)
\textbf{and also}
\( (\text{insert into} \text{ author}(\text{speak, native\_language})\text{ values} \{\text{"Norwegian"}, \text{"English"}, \text{"Norwegian"}) \text{ in\_relationship write with} \text{ p in} \text{ paper where} \text{ p.title = } \text{"The PPP modeling"}) \);

The \textit{insert} statement embeds another \textit{insert} statement, because the entity class \textit{participant} is made up by \textit{author} and \textit{referee}, so any instance of \textit{participant} must also be either an instance of \textit{author} or \textit{referee}, therefore we input the same instance into \textit{author} at the same time. Besides, since the involvement of \textit{author} in the relationship class \textit{write} is a type of \textit{full\_n}, the author must be linked to at least one instance in \textit{paper}, so we also specify a relationship in the statement.

In Example 6 we did not put a value on the attribute \textit{birth\_date} since it is a \textit{private} attribute. In the next example, we use the method \textit{change\_birth\_date} to put an value on the instance.

Example 7:

\textbf{update} participant
\textbf{set value with} change\_birth\_date\( (1963, 10, 29) \)
\textbf{where} \( \text{name} = \{\text{name1} (\text{first\_name: } \text{"Odd"}, \text{middle\_name: } \text{"Ivar"}, \text{last\_name: } \text{"Lindland"}) \}
\)

The formal syntax of SDL is given in the appendix.
3.2.6 What is new in the ONE-R model?

The development of semantic data models boomed in the mid-80s. Researches then gradually turned to more object-oriented approaches. Although there have been many different models proposed, e.g., PM [5], SDM [64], FDM [132], TAXIS [49], IFO [123], etc., they shared a set of common characteristics, especially with respect to data abstraction such as generalization, aggregation, classification and association (grouping), that offer powerful means to describe the structural aspects of objects whose information is to be managed. Although the ONE-R model tries to absorb and exploit the advantages of the individual models in a more unified way, conceptually, its modeling capability can not exceed what the previous models have reached. Naturally, a question rises: what is new with the ONE-R model and, consequently, why do we propose it?

The answer to this question relies on observations of how database technology is applied for information management. First, let’s review a prediction about the development of semantic data model and its application made in 1988 [116]:

"The first research paper on semantic data models appeared approximately 7 years after Codd’s initial publications describing the relational model. Thus, in perhaps another 5-7 years, one of the modeling methodologies discussed here may attain commercial viability."

However, the current situation is quite different from this prediction: the traditional DBMSs are still dominating the applications, whereas there is a strong potential to integrate heterogeneous database into federated databases within a unified conceptual framework [131, 58, 149]. Accordingly, there have been approaches to re-build database schemas in terms of the semantic/OO data models for existing databases [106, 96].

It is clear that for many enterprises where the traditional database techniques have been applied over years, we can not persuade people to give up their databases and re-organize the data with a completely new type of DBMS. They may, however, be persuaded to re-define a data schema on a higher level of abstraction and then establish necessary mappings from this abstraction level to data schemas written in the data languages of different generations of DB techniques.

On the other hand, some semantic data models, e.g., the E-R model, has for a long time been applied as a database design tool, that is, the reality is described in terms of the semantic models, then transformed to the data schemas of a particular DB language such as the relational schema [140]. The problem with this approach is that the operations on the databases are often described in terms of the chosen DB language. A complete understanding of the mapping between the conceptual model to the DB model is therefore needed by the
users of the systems. This can hardly be accepted by naive users in a situation where they have to interact with many different kinds of databases.

<table>
<thead>
<tr>
<th>The concept of &quot;relationship&quot; as an explicit component</th>
<th>IS-A Hierarchy for types</th>
<th>User-defined operations on types</th>
<th>Complex data types</th>
<th>Query Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>SQL-like</td>
</tr>
<tr>
<td>PM</td>
<td>Yes</td>
<td>No</td>
<td>Set-valued</td>
<td>No</td>
</tr>
<tr>
<td>SDM</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>FDM</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Functional query language</td>
</tr>
<tr>
<td>SHM+</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Transaction schema</td>
</tr>
<tr>
<td>TAXIS</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Transaction schema + imperative style programming</td>
</tr>
<tr>
<td>POSTGRES</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>QUEL-like, similar to SQL</td>
</tr>
<tr>
<td>O2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>SQL-like</td>
</tr>
</tbody>
</table>

| ONE-R | Yes | Yes | Yes | Yes  | SQL-like |

Figure 3.20: Comparing ONE-R with other data models on some aspects

The objective of ONE-R is therefore not for developing a new type of DBMS, but have a complete conceptual data language that contains the components for both data definition and data manipulation on a pure conceptual level and is independent of any physical implementation. The concepts and the language are given to all users of databases. On the other hand, since the conceptual language must be implemented on a computerized technical platform, we define an internal form of the data types, so that it can be easily mapped to the available data models. In order to show what we contribute with ONE-R, in Figure 3.20 we compare the features of ONE-R with some other data models.

### 3.3 The Process Port Model

#### 3.3.1 Introduction

PPM, the *Process Port Model*, is an extension of the *data flow diagram* model. The DFD model, since it appeared in the late 1970s [39, 59, 157], has been widely accepted and used to analyze and design information systems with complex functionalities. The major advantage of DFD is that when we begin to analyze a system or part of it, we need only to specify the inputs and outputs
of the processes at a certain level, and ignore the details of the processes. Each process can then be further decomposed for finer analysis until the fully detailed level is reached. This feature is highly in concordance with human beings' way of thinking.

The major weakness of the DFD method is that it is an informal method. A system model is given only with the graphical notations to show the DFD component such as processes, data stores, data flows and external entities without formal expressions. The interpretation of a DFD may be very vague and ambiguous. This makes a formal verification to a system model almost impossible.

This problem has been indicated by one of the inventors of DFD. In a much later published book about the structured analysis [156], Yourdon asked some questions about a DFD such as: In what sequence do the packets of data arrive? How many output items will be produced when a process receives an input item? ...etc. He also gave an explicit answer to the questions:

\[\text{The answer to all these questions is very simple: We don't know. All these questions involve procedural details, the sort of questions that would normally be modeled with flowchart or some other procedural modeling tool. The DFD simply doesn't attempt to address such issue.}\]

The reason for this statement is that: in a DFD we have only incomplete information about the components, whereas a formal verification method needs a fully specified fact base. However, we all know an important rule in software engineering: locate the possible errors as early as possible. This is particularly important for the DFD method because it is a typical stepwise (top-down) method for which a mistake at a high level of description will definitely bring about trouble at the lower levels.

It is obvious that some work should be done in order to get rid of the dilemma. The approaches we would mention in this paper are:

- The Process Port Model (PPM), which was originally developed in the Information Systems Group, and was presented in Berdal and Carlsen's master thesis in 1986 [10]. The modeling method appends to the traditional DFD method more concepts to specify the logical combinations of inputs and outputs of processes, and the control properties of data flows. Afterwards, the model has been developed by other members of the Information Systems Group [125, 110, 135, 108, 61]. A formal verification method based on STD is discussed by Sindre in [135], which is an important source for the contributions given in this thesis.

- The Process Interface Model [85], which was proposed by Kung, a former member of the Information Systems Group.
• The two algebraic approaches by Adler [6] and by Tao and Kung [144] that calculate the possible outputs of a DFD that is the decomposition of a process.

• The Information Engineering methodology [92, 101], where the process dependency relationships among the processes are defined in addition to the data flows.

In the COMIS framework, business functions are described by processes, i.e., the relevant activities that are executed repeatedly. Accordingly, we choose to use the process port model to specify the processes. The illustrations of PPM and its consistency checking method in this section and the next chapter are based on the previous work that is mentioned above, which has been extended to provide

• precise definitions of the conceptual components;

• a canonical form of ports so that consistency checks can be made;

• reuse of process specifications.

3.3.2 Basic concepts

The basic modeling components of PPM, with the graphic notations in Figure 3.21, are illustrated below:

• process A specified activity which can transform some inputs to outputs when it is invoked.

• store A place where a collection of data packages is kept.

• timer A clock which sends out signals of time information regularly.

• agent An external entity which can communicate with the processes, i.e., provide inputs to a process, receive outputs from a processes, or trigger a process. We call it “external” since we are not able to, or do not want to, specify the details of its activities.

• data flow A movement of data. We often just call it flow, because signals may also be delivered between two components, and therefore need to be specified by the concept of “flow”. A flow alway comes from some component (where we call it input flow) and goes into another component (where we call it output flow). When a flow appears at the input or output side of a process, we can further specify the following properties:
3.3. The Process Port Model

a) the basic components of PPP

b) The properties of flows

c) three types of ports
d) a process with composite ports
e) a flow goes into a sink

Figure 3.21: Basic components of PPM
Triggering This notation attached to an input flow of a process means that the input can only arrive when the process is not executing (idle), and the arrivals of the triggering inputs in a legal combination will make the process start an execution. At the beginning of the execution, the process will first receive all the triggering inputs.

Terminating This notation attached to an output flow of a process means that the output can only be sent when the process terminates its execution. All the termination output flows in a legal combination will be sent out before the process changes its state to idle.

Singular Flow Any input/output flow of a process, if not specified particularly, is a singular flow. This means that the flow will be received or sent with only one data package during an execution of the process.

Repeating Flow A repeating input/output flow will be received or sent with data packages more than once during an execution of the process.

Conditional Flow A conditional flow is not mandatory to the process, i.e., during an execution of the process, it may or may not be received in or sent out.

In addition to these basic modeling components, we define an auxiliary concept port which groups some flows with a logical condition showing the logical combinations of the input flows or output flows of a process. An input or output flow itself is a port at the lowest level. Composite ports may be specified of the following types:

- **AND port** All the members of the port (i.e., the lower level ports contained in this port), are going to be received or sent during an execution of the process.

- **XOR port** One and only one member is going to be received or sent during an execution of the process.

- **OR port** At least one member is going to be received or sent during an execution of the process.

Apparently it is a recursive definition, so on a process we can define composite port structures (Figure 3.21d.).

Usually a flow always goes from one component into another one. A possible exception is that the data package in a flow will not be used by any component, i.e., the data package is sent into the flow, then is thrown away. For specifying this case we also define an auxiliary concept sink; all useless flows will be linked to sinks, as that shown in Figure 3.21.e.
Another auxiliary concepts for describing the relations between the input flows and the output flows of a process is the i/o condition. A process "transforms" inputs to outputs. Therefore every output may have some relationships with one or more inputs. In [144], the so called "precedence relations" are specified to indicate that an input "is used to produce" an output. We use the similar ideas, but in a different form.

Figure 3.22: A process \( P \) and its i/o condition

In Figure 3.22 an example is given: the i/o condition of the process \( P \) is expressed by a matrix in which each column corresponds to an input whereas each row corresponds to an output. If any output name appears in a row, then the columns on the row with "X" show a combination of inputs that may be used to produce the output. An output may occupy more than one row to indicate that more than one groups of inputs may be used to produce the output. All rows for one output express the necessary condition for producing the output. For instance, in Figure 3.22, the condition expression for \( o_1 \) is \( (i_1 \wedge i_3) \vee (i_2 \wedge i_3) \), and the condition expression for \( o_2 \) is \( i_1 \wedge i_2 \).

If an output has no row in the matrix, then it is produced unconditionally, i.e., every time when the process is executed, the output may be produced. In Figure 3.22, \( o_3 \) is such an output. For convenience, we can assume that the condition expression for such an output is a disjunction of all the inputs of the process. Therefore, the condition expression of \( o_3 \) is regarded as \( i_1 \vee i_2 \vee i_3 \).

Having defined the modeling components and the auxiliary concepts, now we can define the concepts of process network diagram which is similar to DFD. A PND is a network of processes, stores, timers, agents connected by flows. We call it a process network just because processes are the key components for analysis.

In Figure 3.23 we give an example which models the activities of the organization committee of the IFIP conference that help attendants to book tickets. From the top level PND, we know that:

- process \( P_1 \) may be triggered either by a filled order from an agent (attendant) \( A_1 \), or by his telephone call. However, the two kinds of triggers can not happen at a same time;
Figure 3.23: The activities for ordering tickets in the IFIP conference
3.3. The Process Port Model

- If $P_1$ is triggered by a telephone call, then it will receive items repeatedly from $A_1$ during its execution;

- $P_1$ terminates with a decision for whether the order is unacceptable, and send back the rejected order to $A_1$ or send the acceptable order to a store $S_1$;

- process $P_2$ is triggered by a timer, then it takes the order from $S_1$, book a ticket, then terminates by sending $A_1$ the information about the ticket.

Similarly to the DFD model, a process may be decomposed into a new diagram (process network). The decomposition of $P_1$ is shown in Figure 3.23 which is interpreted as follows:

- process $P_{1,1}$ is triggered by the telephone call and terminates with an empty order;

- process $P_{1,2}$ is triggered by the empty order or an unfinished order sent by itself, receives item information from $A_1$, and fills the item into the order. If the order is unfinished after this, then $P_{1,2}$ terminates by sending the unfinished order which will trigger the process again; else it sends the finished order to $S_{1,1}$ and terminates by sending a request for further processing in $P_{1,4}$;

- process $P_{1,2}$ is triggered by the arrival of a filled order from $A_1$, sends the order to $S_{1,1}$, and terminates by sending a request for further processing in $P_{1,4}$;

- process $P_{1,4}$ is triggered by a request either from $P_{1,2}$ or $P_{1,3}$, takes the order from $S_{1,1}$, terminates by sending out either the rejected order or the accepted order.

From this simple example, we see that more formal descriptions can be given for the processes and the data flows. The auxiliary concepts are not many, but they have provided important information about the behaviors of the processes so that we are better able to analyze a system more precisely, having the logical combinations of inputs and outputs as well as the possible execution sequences of the processes. Moreover, the formal concepts have provided a foundation for consistency check which we will continue to discuss in the remaining part of the thesis.

3.3.3 Substitution of or ports

There is an unlimited number of port structures, but many of them express the same logical combinations of input/output flows. As a simple example, Figure 3.24.a) and Figure 3.24.b) have the same meaning: in any execution of
the processes which have their port structures respectively in the two PNDs, 
either the flow group of a and b, or of a and c, is received or sent, but never both.

Figure 3.24: Two port structures with the same flow combinations

Sindre has noticed this point and suggested some “simplifying rules” [135] in 
order to reduce some redundant parts from a port structure and transform it 
into a clearer one. Our approach takes an alternative: we transform every port 
structure into a unique “normal form”, which is simple and easily understood, 
so that further work for analysis and comparison can be done on this basis.

In order to get to the “normal form” of a port structure, the first step is to 
substitute the or port with other concepts. We are able to do it, because the 
basic concepts introduced in last section have redundancy.

In Figure 3.25 an example with three output ports is shown. The first port 
PortA is an or port with members a and b, while the second port PortB is an 
and port with the same members. These two ports are different, since Port 
A allows either a or b to be sent out whereas Port B requires that both a and 
b are sent out.

Figure 3.25: Comparison of an or port with and ports

On the other hand, Port A and Port B have a common property since both of 
them allow that both a and b may be sent out. The definition of Port B is

Both a and b are going to be sent out.

while the definition of PortA is
3.3. The Process Port Model

At least one of the flows a and b are going to be sent out.

that can also be written as

Both a and b are going to be, but not necessarily, sent out.

The definition implies that both a and b are in fact conditional flows. Therefore, Port C may substitute Port A.

It is apparent that the substitution will permit more flow combinations to be considered legal than that of the original port structure. In the previous example, for instance, the legal combinations of flows for Port A are a, b, and ab whereas the legal combinations for for Port C are a, b, c and none. The last combination of flows for Port C allows that neither a nor b is sent out during an execution, whereas Port A does not allow that situation.

When a substitution allows more legal combinations of flows, some information is lost. The lost information can be partly compensated, however, by static constraints (see Section 4.4). Anyhow, we would rather do the substitutions for all the or ports since a standard port structure with only and and xor ports will offer us great advantages for simplifying the calculations for verifying the consistency of PNDs.

An or port may contain more than two members, where a member may itself be a port. Bases on the analysis of Sinde [135], we substitute all or ports by the following rule:

1. change an or port symbol to that of an and port;
2. mark all flows within the port as conditional flow.

3.3.4 Substitution of xor ports

When there is no or port any more, we can introduce a logical theory and associated theorems:

The Universe of Discourse \( D \): The set of all port names;

The constants and variables \( P_1, P_2, \ldots \);

The Function Themes:

\[ \text{and}(P_1, \ldots, P_n) : D \times \cdots \times D \to D \ (n \geq 1); \]

\[ \text{xor}(P_1, \ldots, P_n) : D \times \cdots \times D \to D \ (n \geq 1); \]
The **Predicates** $Filled(P)$ and $NotUsed(P)$;

The **rules (formulae)** of the theory:

R-I. \( \forall P_1, \ldots, \forall P_n \: Filled(and(P_1, \ldots, P_n)) \Leftrightarrow Filled(P_1) \land \cdots \land Filled(P_n) \)

R-II. \( \forall P_1, \ldots, \forall P_n \: Filled(xor(P_1, \ldots, P_n)) \Leftrightarrow (Filled(P_1) \land NotUsed(P_2) \land \cdots \land NotUsed(P_n)) \lor (NotUsed(P_1) \land Filled(P_2) \land \cdots \land NotUsed(P_n)) \lor \cdots \lor (NotUsed(P_1) \land NotUsed(P_2) \land \cdots \land Filled(P_n)) \)

R-III. \( \forall P_1, \ldots, \forall P_n \: NotUsed(and(P_1, \ldots, P_n)) \Leftrightarrow NotUsed(P_1) \lor \cdots \lor NotUsed(P_n) \)

R-IV. \( \forall P_1, \ldots, \forall P_n \: NotUsed(xor(P_1, \ldots, P_n)) \Leftrightarrow NotUsed(P_1) \land \cdots \land NotUsed(P_n) \)

R-V. \( \forall P_1, \ldots, \forall P_n \: Filled(P) \land NotUsed(P) \Rightarrow NIL \)

R-VI. \( \forall P_1, \ldots, \forall P_n \: Filled(P) \lor NotUsed(P) \)

The semantics of the predicates is to indicate whether a port is to be used to let data packages pass through during the execution of a process. We should mention that

- when data packages have passed through all the members except those conditional flows in an **and** port that consists of conditional flows and other members, it is always assumed that the port has been filled without considering whether data packages have also passed through those conditional flows of the **and** port;

- According to the illustration in Section 3.3.2, when an input port or output port must be filled during an execution of a process, i.e., data packages must pass through it, not all its sub-ports are necessarily to be "filled". For **xor** ports, one and only one lower level port among the members is used to let the data package(s) pass, while all other members must not be used. The properties of composite ports have been described with the rules of the theory.

Now we introduce a definition for specifying that two ports $P_1$ and $P_2$ are **equivalent**:

\[
P_1 \text{ equals } P_2 \equiv_{\text{def}} Filled(P_1) \Leftrightarrow Filled(P_2)
\]

From the theory, we can deduce the following theorems:

**Theorem 3.1:**

\[
\forall P_{11}, \ldots, \forall P_{1m}, \forall P_2, \ldots, \forall P_n \: and(and(P_{11}, \ldots P_{1m}), P_2, \ldots P_n) \\
\text{equals} \\
and(P_{11}, \ldots P_{1m}, P_2, \ldots P_n).
\]
3.3. The Process Port Model

Theorem 3.2:

\[ \forall P_{11}, \ldots, \forall P_{1m}, \forall P_2, \ldots, \forall P_n \text{ xor}(P_{11}, \ldots, P_{1m}, P_2, \ldots, P_n) \]
\[ \text{equals} \]
\[ \text{xor}(P_{11}, \ldots, P_{1m}, P_2, \ldots, P_n). \]

Theorem 3.3:

\[ \forall P_{11}, \ldots, \forall P_{1m}, \forall P_2, \ldots, \forall P_n \text{ and}(\text{xor}(P_{11}, \ldots, P_{1m}), P_2, \ldots, P_n) \]
\[ \text{equals} \]
\[ \text{xor}(\text{and}(P_{11}, P_2, \ldots, P_n), \ldots \text{and}(P_{1m}, P_2, \ldots, P_n)). \]

The proofs are given in the appendix.

As shown in Figure 3.26, the theorems provide some "equivalent" port tree structures. It is also simple to prove that when we substitute a subtree of a port tree with an equivalent subtree, we will get an equivalent port tree. Therefore, the substitution process can be carried out repeatedly, until no further substitution is possible. The substitution algorithm is as follows:

**Algorithm Substitution**

**input:** a port tree with the root node \( P_0 \);

**output:** the changed port tree.

![Figure 3.26: Equivalent port trees](image-url)
begin
if the node $P_0$ is a data flow
then create a new node
\[ P_n = \text{and}(P_0) \]
and let $P_n$ be the root of the port tree
else repeat
begin
search the tree from the node $P_0$;
if the node $P$ has been found that $P$ AND
its children form one of the following structure:
case a: $P = \text{and}(\text{and}(P_{11}, \cdots P_{1m}), P_2, \cdots P_n)$;
case b: $P = \text{zor}(\text{zor}(P_{11}, \cdots P_{1m}), P_2, \cdots P_n)$;
case c: $P = \text{and}(\text{and}(P_{11}, \cdots P_{1m}), P_2, \cdots P_n)$;
then
in case a:
1. eliminate the node and$(P_{11}, \cdots P_{1m})$,
2. change the node $P$ to be and$(P_{11}, \cdots P_{1m}, P_2, \cdots P_n)$;
in case b:
1. eliminate the node zor$(P_{11}, \cdots P_{1m})$,
2. change the node $P$ to be zor$(P_{11}, \cdots P_{1m}, P_2, \cdots P_n)$;
in case c:
1. eliminate the node zor$(P_{11}, \cdots P_{1m})$,
2. create $m$ new nodes
\[ P_{i1} = \text{and}(P_{11}, P_2, \cdots P_n), \]
\[ \cdots, \]
\[ P_{im} = \text{and}(P_{1m}, P_2, \cdots P_n), \]
3. change the node $P$ to be zor$(P_{11}, \cdots, P_{im})$.
end
until the result of the current searching is empty;
end (algorithm)

By applying the algorithm, we are able to transform any port structure to a unique structure which is equivalent to the original port. The unique structure is of the form $\text{xor}(\text{and}(P_{11}, \cdots P_{1m}), \cdots, \text{and}(P_{n1}, \cdots, P_{nm}))$, where every $P_{ij}$ is a data flow. We call the port canonical port because the structure is unique and clear, like that shown in Figure 3.27.a).

![a). The canonical port structure](image1.png)
![b) The canonical port structure expressed by another kind of graphical notations](image2.png)

Figure 3.27: The equivalent port structure for a process $P$

In Figure 3.27.b), we use another graphical notation to express the canonical
port structure. At the input side of the process, there is one or more canonical input ports (CIPs) while there is one or more canonical output ports (COPs) at the output side. A canonical port is just like a terminal linking the process with its environment. During an execution of the process, only one CIP and one COP are used. In addition, a data flow can flow into or flow out from different ports in the different executions of the same process, like the example given in Figure 3.24.b). In Figure 3.27.b), this feature is shown more intuitively.

Corresponding to the canonical port structure, we also refine the component data flow by regarding it as consisting of three smaller parts output, flow-pipe and input:

- an output delivers one of the result data of a process to its outside;
- a flow-pipe is a channel which lets data pass through it. A flow-pipe has its capacity, i.e., the maximum number of data the pipe can contain at any specific time.
- an input receives one of the resource data into a process.

![Diagram](image)

Figure 3.28: Data flows consisting of flow-pipe and input/output

In figure 3.28 we show some data flows under different conditions. We can see that a data flow from process to process consists of all the three parts, while a data flow between a process and a data store, as well as between timer and agent, only consists of a flow-pipe and an input (or an output). Further more, we can see that an input can receive data from more than one flow-pipe as well as that an output can put data into several flow-pipes simultaneously.

Having refined the definition of data flow, we concentrate on the inputs and outputs of a process when we analyze it. The properties of data flows, i.e.,
triggering, terminating, singular, repeating, conditional, will only be given to inputs and outputs of the processes. Therefore, we can let the flow-pipe inherit the name of the data flow while we regard the output and input as the essential parts of the processes. We call a PND which is drawn with canonical ports and the refined data flows canonical process network diagram. In Figure 3.29 we draw the canonical PNDs of the IFIP activities for booking tickets shown in Figure 3.23.

I shall explain on why we need to define the canonical port structure as well as the refined definition for data flows. The first motivation is from the consideration for obtaining consistency check. One of the initial objectives of PPM is to let the auxiliary concepts help to build correct system models. A reasonable way to reach the goal is to transform a port structure into a unique form that is equivalent to the original one, but where its properties are easier to check and compare with that of another port structure. The canonical port structure is such a unique form; the benefits we get from it for consistency check will seen in the next chapter.

Another motivation is from the consideration for reuse of process specifications. This topic is discussed in the next sub-section; here we only briefly describe the problem:

- In ordinary DFD or PND, a data flow is an independent component with a unique name. However, if a process is reusable, then it may receive or send data flows with different names in different diagrams. Therefore, it is more reasonable to give unique names to the inputs and outputs of the process and regard them as unseparable parts of the process;

- A reusable process may be able to receive or send some inputs and outputs with xor combinations, say, A xor B, then when the process appears in different diagrams, either A or B is a legal input or output. This means that a reusable process may appear in some process networks with only part of its inputs and outputs.

With the canonical port structure and the refined flow definition, it is much easier to deal with these problems.

Finally, we must answer the question: is the canonical structures purely an internal form, or can it also be used by the users (i.e., the builders of system models)? The answer is that, as long as the possibility of defining reusable processes exists, the users must be able, but are not forced, to use both the original and canonical forms of port structure as well as that of the data flows. Since the semantics is clear, they can mix two kinds of graphical notations in one diagram. All of these will internally be transformed into the canonical form.
Figure 3.29: The canonical PNDs of IFIP ticket booking activities
3.3.5 Reusable definitions

Reuse, as an important means to enhance software and system development productivity, has been studied and applied widely [44, 105, 54]. Generally, the basic modeling concepts of reusable system components are the primitive components of systems specifications such as object types, processes, stores, external entities, and so forth [101, 18, 9], but there are also approaches to treat even a whole specification, say, a DFD, as the unit of reuse [95, 139].

Like SA and IE, in PPM we let the component process be main object of reuse. A process $P$ may be a part of many different activities, so it may appear in many PNDs. However, we need only to specify its detail, i.e., the decomposition of it, at one place and let its appearance at other places just be a reference to it, namely $P'$. The case is shown in Figure 3.30 together with the graphical notations for a reusable process and the reference to it.

![Diagram showing process $P$ and $P'$ with their graphical notations]

Figure 3.30: Reuse of a process and the graphical representations

Actually, Figure 3.30 shows a reusable component within the scope of a business system, i.e., the definition of the reusable process can be found in a process hierarchy which itself is a part of a business function. However, there also exist some “general purpose processes” which can appear in many business systems, e.g., generating a report, sending out an acknowledge letter, etc. The specifications of these processes should be outside of any particular business functions. In PPM, we support the definition of such kind of process as shown in Figure 3.31:

- A general purpose process is defined independent of any specific business function, so only its inputs, outputs and the corresponding port structures and I/O condition are specified;

- The reference to such a process may use parts of the inputs and outputs, if the used part is “compatible” with the definition of the process, i.e., the canonical form of the part has at least one CIP (Canonical input port) and one COP (Canonical output port), and the I/O condition of the process allows the combination of the CIP(s) and COP(s).
Another problem is how to let the flows carry the "control information". We know that in the process model of IE, the "dependency relationships" among processes are to be explicitly specified [101] whereas in PPM the knowledge is expressed by data flows with triggering and/or terminating properties. However, in the definition of a general purpose process one may not have information on which process will trigger it or which process it will trigger. When it is put into a specific PND, how to show the dependency relationships between this process and other processes?

Our solution to the problem is:

- If data flows from some other processes to the referenced process and the flows from the referenced process to other processes can express the control properties, then do nothing more than the original specification of a PND;

- In the case that the flows to the referenced process or from it can not express the control properties, we augment the reference by creating an and port at either the input side or at the output side, or at the both sides when needed. The augmented port contains an extra flow and has the property of triggering or terminating while the original inputs or outputs have lost such properties. An extra flow can only be a signal to convey the control of the execution of processes.

- On an augmented reference to a process, the contents of the I/O condition are kept while a new term is added in order to show that the new termination flow depends on the triggering flow(s).

An example is shown in Figure 3.32. A reusable process \( P \) has input \( i_1 \) and outputs either \( o_1 \) or \( o_2 \) (Figure 3.32.a). When it is put to a PND, it should be executed after the process \( P_1 \) terminates and then get input from a data store.
$S_1$(Figure 3.32.b). When the execution of $P$ terminates, it sends output to either the data store $S_2$ or $S_3$, and then lets process $P_2$ execute consequently. 

The control information is carried by the flows $f_a$ and $f_a'$ that have been put into the augmented input and output and ports.

![Diagram of a general purpose process $P$](image)

**Figure 3.32:** A reusable process and its augmented reference

### 3.3.6 More auxiliary concepts for other components

In the previous sections, we concentrated on the properties of processes because the process concept is the most important component in PPM. However, more auxiliary concepts should be given in order to easier specify the behaviors of the modeling components.

Having borrowed the ideas of *behavior rules* from RUBRIC and TEMPORA [12, 13], we use *triggering rules* to specify the behavior of an agent or a timer. The form of triggering rule is:

```
When event
[If pre-condition ]
Then send flow1[, flow2...];
```

In the above statement, *event* and *pre-condition* are all boolean expressions, but the truth of an event will hold only a instantaneous moment. It means
that when an event happens, if some conditions are true, then an agent or a timer sends data into one or more data flows which then trigger the execution of a process.

An agent is assumed to be more "intelligent" than a timer, so the event and pre-condition parts of the triggering rule of an agent may contain statement about time information, state of data stores and processes, or even the state of the agent himself. However, since a timer is only a clock, its capability for determining the truth of a condition is limited. Therefore, the events that appear in the triggering rules of timers are only time events, and the pre-condition part of the triggering rules can only indicate the work state of processes. For instance, the triggering condition of $T_1$ in Figure 3.23 is defined as:

\begin{itemize}
  \item [When] time is 8:30, 10:30, 12:30 or 14:30
  \item [If] $P_2$ is idle
  \item [Then] send $f_6$
\end{itemize}

This means that $T_1$ will send a time signal to trigger $P_2$ four times a day. Each time, if the previous execution of $P_2$ has terminated, then the signal is sent and $P_2$ is triggered.

Another auxiliary concept is the flowing manner that indicates how a flow goes from a data store. No doubt a flow going into a store will put a new datum into the store; however, the flow from a data store may really bring away a datum (consumes it), or may contain just a copy of the datum. In the general case, we consider the second manner as default, and allow the first manner to be specified explicitly. In Figure 3.33, the graphic notations are given.

\begin{figure}[h]
\centering
\includegraphics[scale=0.5]{flow_manner.png}
\caption{Two flowing manners from data stores}
\end{figure}

### 3.3.7 The cross references between PPM and ONE-R

Within the integrated modeling framework, ONE-R is used to specify the types of information to be managed (operands) whereas PPM is used to specify the
information processing activities (operations). In order to further specify the relationships between operands and operations, cross references between the two models must be built.

![Diagram](image)

External scenarios

Figure 3.34: Cross references between PPM and ONE-R

The first step to build the cross reference, as that is done in SSADM [29], is to indicate what types of data are contained in data stores. We will link every data store with an external scenario in ONE-R, and indicate what data items in the scenario are to be stored, as shown in Figure 3.34 where some parts of a scenario is marked. The second step is to indicate what types of data to be processed in a process by linking the process with an external scenario which contains all the items that are stored in all the data stores which is accessed by the process. This is also shown in Figure 3.34.

It should be noticed that a general purpose reusable process may deal with different data items when it is used in different PNDs, thus it is probably meaningless to say what is the cross reference between such a process and a scenario; instead, it is meaningful to build the cross reference between a process reference and a scenario. Hence, we may not build any cross reference for a general purpose process but just build various cross reference for the different references to the process.

Because ONE-R and PPM are relatively independent of each other, it is not reasonable to require that the external scenarios needed by all processes are defined within the context of ONE-R modeling activities. Only the “basic” scenarios can be built without having insight into the context of processing activities. All the necessary external scenarios may be defined one by one along with the progress of specifying the process hierarchies. Therefore, the formation of an ONE-R model and a process-port model is an interactive process shifting between the data-oriented and process oriented perspectives.
3.4 The Actor Model

3.4.1 Introduction

The Actor Model is used to specify executable programs. The concept actor comes from the object-oriented paradigm in which an object may be a component of a program, a database record, or even a whole program [105, 138, 51]. In AM, an actor is also an object, but the behavior of it only shows the main functions of a program as a whole. The program itself may also contain some smaller objects, especially when the program is written in an object-oriented language. However, those will only be considered in the detailed design phase and are not within AM’s context. AM will be used to specify the functions of programs written in any programming language.

When the object-oriented paradigm was expanded from programming languages to wider scopes like software systems, databases and conceptual modeling of information systems, there were two styles in dealing with the relationship between the static (structural) and dynamic (behavior) aspects of objects. In the first style one insists on dealing with the two aspects in a unified way, even though the difference between active objects and passive objects are recognized [130, 51]; in the second style, one makes clear distinctions between active and passive objects in order to distinguish subjects who perform actions from objects which are acted upon [50, 134, 40]. This viewpoint is also supported by the FRISCO report in which the concepts of “actor” and “entity” are clearly distinguished [45].

We support the second viewpoint because it is closer to our understanding of information processing. For instance, a clerk in a sales department deals with the information about customers, while the information about him is dealt with by another clerk in the personnel department. So in a business system model, there should be two object(or entity) types about the clerk: the first one represents an active role whereas the second one represents a passive role. By the same principle, the behavior(functions) of programs have nothing to do with their structural aspects such as module structure, calling relationship, etc.

There has been a confusion in the interpretation of the meaning of the terms “actor” and “agent”. The two concepts are often used with the same or similar meaning, e.g,

- “An actor is a conception of a phenomenon in an OS (Open System) that is seen as carrying out and controlling the execution of activities in an OS” [45].

- “The agent concept defines the role played by individual office workers, or groups of workers, when they interact with the OISs (Office Information
Systems). An agent may perform, or be responsible for, an activity” [50].

- “Agents may be humans, humans interacting with computers, humans working with computer support, and computer systems performing tasks without human intervention” [16].

Therefore, we must give a clear definition of the two concepts since both of them are used in the COMIS modeling framework. In Figure 3.35, the is-a hierarchy of object types in the COMIS framework is shown with the following illustrations:

- the concept of object is specialized into that of passive object and active object. A object set can correspondingly be partitioned into two sub-sets for active objects and passive objects. The passive objects are specified by the ONE-R modeling.

- the concept of active object is further specialized into that of agent and actor. Both will be used in PPM. But, particularly, actors will also be specified by AM modeling.

- The concept of agent is used to represent humans whose behaviors are sometimes uncertain, whereas the concept of actor is used to represent computer programs whose behaviors are determined.

Why do we need AM? The requirement arises since we must be able to describe how the information processing activities in a business system are supported by technical systems. Under the current conditions of technology, there are two sources of such technical systems. The first source is the software market, just as said in [50]:
3.4. The Actor Model

The OIS design process primarily tries to combine and interface software components available on the market, rather than to develop new packages from scratch. Software development is limited to providing new functionalities not available on the market, to tailoring and extending software packages, and to configuring selected components for use in the procedures of the OIS under design.

The second source, as stated above, is to develop systems that are not available on the market.

Both sides need a modeling tool to specify what roles a software component (program) plays in supporting the processes and how the process specifications are grouped to form new technical systems. AM is just such a modeling tool.

3.4.2 The modeling concepts

The basic construct in AM is actor. The actor concept represents an executable program. The properties of an actor are abstracted by two more concepts, function and working state. In order to explain the concepts, we give an example in Figure 3.36 which illustrates a program “report generator”. From the example we know:

![Diagram of the model of an actor]

Figure 3.36: The model of an actor
an actor has one or more functions, each of which may receive some input(s) and produce some output(s) upon the input(s). Since the execution of a function may be viewed as a process transforming some input(s) to output(s), we use the same graphical notations for functions as that for processes. For the same reason, we also use I/O conditions to describe the relationships between inputs and outputs of the functions.

A working state contains, when it is not empty, some data items (variables) which describe some aspects of the working state of the actor. For instance, in Figure 3.36, the working state contains an item to describe which default printer is to be used when producing a report. Similar descriptions can be found for many programs, e.g., The Xdefaults for the X window system.

Apparently, the above definition is analogous to that for the concept object in the object-oriented paradigm, since an object also groups some functions (or procedures, methods, etc.) onto itself. The difference is that the working state of an actor has almost nothing to do with the actor's structural aspects, because an active object mainly performs operations on other objects. Therefore, when we specify a program as an actor, we need not to be concerned about its structural properties, e.g., the module structure of the program, but some only be concerned with the variables concerning the execution status of the program.

By the way, the concept of actor is just an abstraction of an executable program. A program itself may have more properties and aspects to be further specified, e.g., user interfaces, objects (if the program is written in an OOPL), etc., but we ignore all these details when we use AM to specify a program except the basic functionalities of it with the legal input/output combinations. This way of abstraction will provide us with strong means to describe the relationship between a business system and the supporting technical systems.

As shown in Figure 3.36, in addition to the port structures that show the legal combinations of inputs and outputs of the functions, the concepts of triggering and terminating have also been applied to the inputs and outputs. However, the meaning of the properties differs from PPM: A triggering input does not really "trigger" an execution of the actor, but has already contained a data item when the actor is invoked; similarly, a terminating output does not really "terminate" the execution but the data item of it is produced just when the execution terminates. The non-triggering inputs and non-terminating outputs access requested data during the execution.

3.4.3 The mapping between AM and PPM

In our modeling framework shown in Figure 3.3, the actors of an application system can be modeled by both PPM and AM. This means that a mapping
exists between the AM and PPM when an actor is also described in terms of PPM.

![Diagram](image)

**INPUTS / OUTPUTS**

- I1 (I11) : report spec.
- I2 (I21) : report format file
- I3 (I22) : data set
- I4 (I23) : printer name
- O1 (O11) : report format file
- O2 (O12) : error message file
- O3 (O21) : generated report

**I/O CONDITION**

<table>
<thead>
<tr>
<th></th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O3</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Figure 3.37: Mapping the actor into a reusable process (the process is drawn by two kinds of graphical notations)

![Diagram](image)

**Figures 3.37, 3.38**: Grouping processes into an actor
An example is given in Figure 3.37 where the actor "report generator" in Figure 3.36 is mapped to a reusable process by the following transformation rules:

1. The actor itself is mapped into a reusable process;
2. The process has an xor input port whose members are the input ports of all the functions of the actor; in the same way, the process has an xor output port as well;
3. The inputs and outputs need to be renamed and, when necessary, some inputs or outputs should be merged;
4. the I/O conditions of the functions are also merged with the corresponding changes of names (of inputs and outputs).

Having made the mapping, we may then can regard the actor as a "general purpose" process. Whenever a function of the actor can support a process, we can replace the process in a PND by a reference to the general process with the inputs and outputs of the function. According to the mapping rules, the inputs and outputs must be a "compatible" part of the general purpose process.

On the other hand, if some processes need technical support but the market available software products can not support them, then we can also "group" several processes into an actor, as shown in Figure 3.38. The grouping is actually a mapping, but in the opposite direction. The processes will automatically become the functions of the actor to be built, and the new working state may contain the variables for describing the process logics (see that illustrated for DFDs [156]). After the actor is built, or at least is planned, we can map it again to a general purpose process, and replace all the grouped processes in the PNDs where they appear by the corresponding references to the reusable process.

3.4.4 The cross references between AM and ONE-R

We will not build cross reference between an actor and a scenario, since when an actor executes in different places it may access different data items. Instead, we build the indirect cross references, say, the cross references between scenarios and the process references to the functions of an actor. The mapping has been illustrated in the previous section and Figure 3.34. By the way, we will be able to specify the "external data schema" of the actor in a information processing activity, and the schema is just the union of the scenarios linked to the process references of the actor.
3.5 Brief Summary

Returning to Figure 3.3 which shows our modeling framework, we can now claim that the modeling framework is supported by the three models and the cross references among them. By the way, a business system and its underlying technical systems, either available in the market or to be developed, can be specified with data, process and behavior perspectives:

- The types of data to be managed in a business system are described by ONE-R with the scenarios, each of which contains some related entities;
- The information processing activities are grouped into business functions; each business function is a PND containing some “top” level processes; each process is triggered by agents or timers according to their behavior rules;
- A process may be further decomposed into a lower level PND; the decompositions can be further done to a bottom level;
- an application system can be a collection of actors; each actor has a set of functions and can me mapped to a reusable process, so that the mapping between PPM and AM shows how a technical systems support the activities of a business system;
- The data items to be used by the processes and supporting technical systems are described in external scenarios which in turn have mappings to the basic scenarios.
Chapter 4

Consistency verification

One of the basic requirements to an information system specification is its correctness. In other words, there should not be conflicts within the descriptions of the properties of the system. Only if the correctness of the specification can be formally proven, can one be absolutely convinced that a correct system model has been built. In this chapter we address this topic with our modeling approach.

Since the proofs of the correctness of system specifications must be mathematical proofs, we use the word "consistency" to replace "correctness". In mathematical logic, a theory, i.e., a set of formulae written in a logical language, is consistent if and only if there is no conflict implied by the theory. The definition is sufficient for our purpose, if we treat the system specification as a set of logical formulae.

Hence, we need formal specifications as the basis of consistency verification. There are two approaches:

- the algebraic specification approach specify a system as a set of abstract data types (ADTs). The theory of an ADT consists of a set of symbols (sorts and operations) (signature) and a collection of formulae (the axioms of the theory); the interpretation of the theory as a many sorted algebra [62, 155, 14, 34]. The specification is a set of theories and the relationships between them. This approach has been applied in the object-oriented system paradigm where a system is regarded as a society of interacting objects [2, 154, 19, 130, 51, 117];

- the logical theory approach, on the other side, treat the schema of a database and the associated operations on it as ONE logical theory. This approach has also been widely applied [79, 63, 121, 109].

Both approaches have advantages, but we prefer the logical one for our modeling framework because the logical view supports the concept of global state
of a database [128]. In addition, Kung [83] has provided a set of methods for consistency verifications on information system specifications including static constraints, operation specifications and temporal constraints along with the algorithms to check if the specification is decidable with respect to its consistency.

We still can not expect to express all of the concepts of a system model as one logical theory and use the logical inference mechanism to verify the consistency of the system model. Such an attempt will be impracticable or even impossible:

- Logical languages are not user-friendly and have no structures which are easily used for information systems. Even in Kung’s pure logical approach of [82], it is still assumed to “use a separate modeling approach for the modeling task and then specify the conceptual model in terms of a logic oriented language”.

- Since we propose an integrated modeling framework which consists of several sub-models, the concepts and the relationships between them may be too many to be easily organized in one logical theory;

- It is difficult to check the process model by a purely logical method, because we start the modeling by specifying the processes with incomplete knowledge, say, only knowledge about inputs and outputs. When a process is decomposed into a set of smaller processes, the components still are “black boxes” and their internal structures are often unknown even though we must decide if the composition is correct.

Under these considerations, we redefine the following concept of “consistency” within our modeling framework:

- completeness – a model for a business system and its supporting technical systems contains all the necessary conceptual components;

- consistency – there is no conflict within the descriptions of scenarios, i.e., when DB schema consisting of basic scenarios is translated into a logical theory, the theory is consistent. The principle is also applied to the description of processes and actors.

- constructivity – the decomposition of a process into a process network diagram (PND) will preserve the external properties of the process, i.e., the PND will use the same inputs and produce the same outputs as that of the process.

The above definitions divide the task of consistency check & verification into several parts, for each of them we will provide a method to deal with the problem of consistency. It is worth mentioning that state-transition-diagrams (STDs) are used in addition to the logical inference methods, since we have
stated that purely logical methods can hardly solve all of the problems that we have to deal with.

4.1 Transforming an ONE-R DB schema into a logical theory

In this section, we will solve the problem of transforming an ONE-R scenario into a logical theory.

The transformation, in some sense, depends on the form of data schemas. The work of Jacobs [77] shows that a database schema form may result in a set of second order logical formulae. However, the work of Rybinski [122] shows that the data schema of three database types—relational, hierarchical and CODASYL-like, can all be transformed into first order logical formulae. Although ONE-R is a data model with complex data types, we are also able to get a set of first order formulae since in the previous chapter we have developed the formal expressions of ONE-R with a four-leveled type structure.

In the section for ONE-R, we defined a data base schema as a 6-tuple

\[(T \text{set}, TD \text{set}, D \text{set}, A \text{set}, \text{dom}, C)\]

where \(T \text{set}\) contains a set of type names. More information about the types is given in the other formal expressions, particularly the type constructors. On that basis, we may first define a many-sorted and first order logical language as follows:

1. The sorts for terms (to be defined later):
   - For each \(T \in T \text{set}\) there is a symbol \(S_T\); any term \(t\) is linked to a sort symbol, saying that \(t\) is of sort \(S_T\);
   - if there is an expression \(E = IS.A(T', T)\) and \(Id.of.T'\) and \(Id.of.T\) are ID types of \(T'\) and \(T\) respectively, then \(S_{Id.of.T'}\) is compatible with \(S_{Id.of.T}\);
   - For any \(T, S_T\) is also compatible with \(S_T\) itself;
   - For any \(T_1, T_2, T_3\), if \(T_1\) is compatible with \(T_2\) and \(T_2\) is compatible with \(T_3\), then \(T_1\) is also compatible with \(T_3\).

2. The alphabet set:
   - The sets of constant symbols, including a set \(\{c_T^{(1)}, \ldots c_T^{(m_T)}\}\) for each \(T \in T \text{set}\). \(c_T^{(1)}, \ldots c_T^{(m_T)}\) are of sort \(S_T\);
   - The sets of variable symbols, including a set \(\{x_T^{(1)}, \ldots x_T^{(n_T)}\}\) for each \(T \in T \text{set}\). \(x_T^{(1)}, \ldots x_T^{(n_T)}\) are of sort \(S_T\);
• The set of connective symbols \{\land, \lor, \neg, \rightarrow\};

• The punctuation symbols \(\",\", (\" and \")\);

• Quantifiers \(\forall\) and \(\exists\);

• The set of predicate and function symbols containing:

  - a set of built-in binary predicates
    \[\{<,\leq,>,\geq,=,\neq,\subseteq,\supseteq,\subset,\supset,\in,\ni\}\]

    In using these predicates to express some relations, we follow the conventions to write the expressions, for example, use always use \(x_1 = x_2\) rather than the expression \(= (x_1, x_2)\) which actually follows the grammar.

    For the predicates \(<,\leq,>,\geq,=,\neq\), the two terms as their arguments must be of the same sort or of two sorts one of that is compatible with the other.

    For the predicates \(\subseteq,\supseteq\), the argument as an element of another argument must be of the sort that is compatible with the sort of the underlying type from which the class type is constructed.

  - for any type \(T \in Tset\), we may have following predicates and functions:

    If \(\text{CONS}_T = (A_1 : T_1, \ldots, A_n : T_n)\), then for each \(A_i(i = 1, \ldots, n)\), there is a predicate

    \(\text{Att}_{TA_i}\) (binary, the first argument is of sort \(S_T\) and the second argument is compatible with \(S_{T_i}\))

    and a function

    \(\text{att}_{TA_i}\) (binary, the first argument is of sort \(S_T\) and the second argument is compatible with \(S_{T_i}\)) which is of sort \(S_{T_i}\);

    If \(\text{CONS}_T = \bigcup(A_1 : T_1 \ldots A_n : T_n)\), then for each \(A_i(i = 1, \ldots n)\), there is a predicate

    \(\text{Alt}_{TA_i}\) (binary, the first argument is of sort \(S_T\) and the second argument is compatible with \(S_{T_i}\))

    and a function

    \(\text{alt}_{TA_i}\) (binary, the first argument is of sort \(S_T\) and the second argument is compatible with \(S_{T_i}\)) which is of sort \(S_{T_i}\);

    If \(\text{CONS}_T = \{T\}(m1 : m2)\), then there is a function

    \(\text{card}\) (unary, the argument is of sort \(S_T\)) which is of sort \(\text{Sinteger}\);

    If \(\text{CONS}_T = (T')\), then for any predicate \(P'\) or function \(F'\) defined on \(T'\), there is also a predicate \(P\) or function \(F\) which is obtained by substituting the symbol \(T'\) with \(T\) in \(P'\) or \(F'\), and any argument required to be of sort \(S_{T'}\) or compatible with it in \(P'\) and \(F'\) is required to be of sort \(S_T\) or compatible with it in \(P\) and \(F\);

    If \(T\) is an entity type with the ID type \(ID_T\), then there is a predicate
4.1. Transforming an ONE-R DB schema into a logical theory 121

\[ Id_{ofT} \] (binary, the first argument is of sort \( S_T \) and the second argument is compatible with \( S_{ID_T} \))

and a function

\[ entity_{ofID_T} \] (unary, the argument is compatible with \( S_{ID.T} \))

which is of sort \( S_T \);

If \( T \) is an \( n \)-nary relationship type with the participating types \( T_1, \ldots, T_n \) and their ID types \( ID_1, \ldots, ID_n \), then there is a predicate

\[ Ids_{ofT} \] ((\( n+1 \))-ary, the first argument is of sort \( S_T \) and the \( 2_{nd}, \ldots, (n+1)_{th} \) arguments are compatible with \( S_{ID_1}, \ldots, S_{ID_n} \) respectively)

a function

\[ relationship_{of\{ID_1, \ldots, ID_n\}.n.T} \] (\( n \)-ary, the \( 1_{st}, \ldots, n_{th} \) arguments are compatible with \( S_{ID_1}, \ldots, S_{ID_n} \) respectively) which is of sort \( S_T \)

and a predicate

\[ Related_T \] (\( n \)-ary, the \( 1_{st}, \ldots, n_{th} \) arguments are compatible with \( S_{T_1}, \ldots, S_{T_n} \) respectively);

- If there is any expression of the form \( IS.A(T', T) \) in \( C \) and \( ID_{T'} \) and \( ID_T \) are ID types of \( T' \) and \( T \) respectively, then there is a predicate

\[ IS.A \] (binary, the first argument is compatible with \( S_{T'} \) and the second argument is of sort \( S_T \)).

3. The grammar of the language:

- The rules to form the terms of the language:

  1). A variable is a term;
  2). A constant is a term;
  3). If \( f \) is an \( n \)-ary function symbol and \( t_1, \ldots, t_n \) are all terms that meet the sort requirement for the \( i_{th} \) argument of the function \( f \) respectively (\( i = 1, \ldots, n \)), then \( f(t_1, \ldots, t_n) \) is a term;
  4). A term can only be formed by finite number of applications of the above rules.

- The rules to form the well-defined formulae (wffs) of the language:

  1). If \( P \) is a \( n \)-ary predicate symbol and \( t_1, \ldots, t_n \) are all terms that meet the sort requirement for the \( i_{th} \) argument of the predicate \( P \) respectively (\( i = 1, \ldots, n \)), then \( P(t_1, \ldots, t_n) \) is a wff and we call this form of wffs atomic formula;
  2). If \( f \) is a wff, then \( (w) \) is a wff;
  3). If \( w \) is a wff, then \( \sim w \) is a wff;
  4). If \( w_1 \) and \( w_2 \) are all wffs, then \( w_1 \wedge w_2, w_1 \vee w_2 \) and \( w_1 \rightarrow w_2 \) are all wffs.
5). If \( w \) is a wff and \( x_T \) is a variable, then \( \forall x_T w \) and \( \exists x_T w \) are all wffs;

6). A wff can only be formed by finite number of applications of the above rules.

Based on these definitions, we recall the simplifications we introduced in Section 3.2.4:

- \( x.A_i \cdots A_k \) may be a term when \( x \) is a variable or constant of a tuple type \( T \). This term is a substitution to a set of functional mappings to \( x \);

- The expression \( A_i \cdots A_k \{ y \} \) can be used to substitute the expression \( y \in A_i \cdots A_k \).

Then in terms of the language, we can transform the DB schema into a logical theory, i.e., a collection of formulae formed in following ways:

- for any \( T \in \text{Tset} \), we may have following formula:

  If \( CONS_T = (A_1 : T_1 \cdots A_n : T_n) \), then for each \( A_i (i = 1, \cdots n) \), there is a formula

  \[
  \forall x_T^{(1)} \exists x_T^{(1)} (\text{Att}_{TA_i}(x_T^{(1)}, x_T^{(1)})) \iff \text{att}_{TA_i}(x_T^{(1)}) = x_T^{(1)}
  \]

  when \( T_i \neq T^* \) (null value is not allowed), there is a formula

  \[
  \forall x_T^{(1)} \exists x_T^{(1)} (\text{Att}_{TA_i}(x_T^{(1)}, x_T^{(1)}))
  \]

  If \( CONS_T = \bigcup (A_1 : T_1 \cdots A_n : T_n) \), then there are formulae

  \[
  \forall x_T^{(1)} \exists x_T^{(1)} (\text{Alt}_{TA_i}(x_T^{(1)}, x_T^{(1)})) \iff \text{alt}_{TA_i}(x_T^{(1)}) = x_T^{(1)}
  \]

  and

  \[
  \forall x_T^{(1)} ((\exists x_T^{(1)} (\text{Alt}_{TA_1}(x_T^{(1)}, x_T^{(1)))) \vee \cdots \vee (\exists x_T^{(1)} (\text{Alt}_{TA_n}(x_T^{(1)}, x_T^{(1))))))
  \]

  If \( CONS_T = \{ T' \} (m_1 : m_2) \) then if \( m_2 \) is a number, then there is a formula

  \[
  \forall x_T^{(1)} ((\text{card}(x_T^{(1)}) \geq m_1) \land (\text{card}(x_T^{(1)}) \leq m_2))
  \]

  else there is a formula

  \[
  \forall x_T^{(1)} (\text{card}(x_T^{(1)}) \geq m_1)
  \]

  If \( CONS_T = \{ T' \} \) there for any formula \( W' \) defined on \( T' \), there is a similar formula \( W \) which is same as \( W' \) except that the symbol \( T' \) in \( W' \) is substituted with \( T \).

  If \( T \) is an entity type with ID type \( T' \), then there is a formula

  \[
  \forall x_T^{(1)} \forall x_T^{(1)} (\text{Id.of}(x_T^{(1)}, x_T^{(1)})) \iff (\text{entity.of}(x_T^{(1)}) = x_T^{(1)})
  \]
4.1. Transforming an ONE-R DB schema into a logical theory

If $T$ is a $n$-ary relationship type with participating types $T_1, \ldots, T_n$ and their ID types $ID_1, \ldots, ID_n$, then there is a formula

$$
\forall x_T^{(1)} \forall x_{ID_1}^{(1)} \ldots \forall x_{ID_n}^{(1)} (Ids \cdot of(x_T^{(1)}, x_{ID_1}^{(1)}, \ldots, x_{ID_n}^{(1)})) \leftrightarrow (relationship \cdot of_{(ID_1, \ldots, ID_n, in, T)}(x_{ID_1}^{(1)}, \ldots, x_{ID_n}^{(1)}) = x_T^{(1)}))
$$

- For any expression $E$ in $C$, we may have following formulae:

  If $E = IS \cdot A(T', T)$ and $CONS_T = (A_1 : T_1, \ldots, A_n : T_n)$, then there are formulae

  $$
  \forall x_T^{(1)} \exists x_T^{(1)} IS \cdot A(x_T', x_T)
  $$

  $$
  \forall x_T^{(1)} \forall x_T^{(1)} (IS \cdot A(x_T', x_T) \rightarrow ((\text{att}_{TI}(x_T^{(1)}) = \text{att}_{TI}(x_T^{(1)}))) \land \\
  (\forall x_T^{(1)} (\text{Att}_{TI}(x_T^{(1)}, x_T^{(1)}) \leftrightarrow (\text{Att}_{TI}(x_T^{(1)}, x_T^{(1)))))) \land \ldots \land \\
  (\forall x_T^{(1)} (\text{Att}_{TI}(x_T^{(1)}, x_T^{(1)}) \leftrightarrow (\text{Att}_{TI}(x_T^{(1)}, x_T^{(1)))))) \land \\
  )
  $$

  and

  $$
  \forall x_T^{(1)} \forall x_T^{(1)} \forall x_T^{(1)} ((IS \cdot A(x_T', x_T) \land Related_R(x_T^{(1)}, \ldots, x_T^{(1)})) \rightarrow Related_R(x_T^{(1)}, \ldots, x_T^{(1)}))
  $$

  where $R$ is a relationship type in which $T$ is involved and $R$ is the augmented relationship type of $R$.

  If $E = DISTINCT(T_1, \ldots, T_n)$, then for any pair $i, j$ ($i = 1, \ldots, n$, $j = 1, \ldots, n$, $i \neq j$), there is a formula

  $$
  \forall x_{T_i}^{(1)} \forall x_{T_i}^{(1)} (\text{att}_{TI}(x_{T_i}^{(1)}) \neq \text{att}_{TI}(x_{T_i}^{(1)}))
  $$

  If $E = MAKING \cdot UP(T_1, \ldots, T_n, T)$, then there is a formula

  $$
  \forall x_T^{(1)} (\exists x_T^{(1)} (\text{att}_{TI}(x_T^{(1)}) = \text{att}_{TI}(x_T^{(1)})))
  $$

  $$
  \lor \lor \lor (\exists x_T^{(1)} (\text{att}_{TI}(x_T^{(1)}) = \text{att}_{TI}(x_T^{(1)})))
  $$

  If $E = TYPE \cdot OF \cdot RELATIONSHIP(R, P_1, \ldots, P_n)$ and the participant types of $R$ are $T_1, \ldots, T_n$, then for any $i = 1, \ldots, n$, if $P_i$ is $full \cdot 1$ or $full \cdot n$, then there is a formula

  $$
  \forall x_{T_i}^{(1)} \exists x_{T_i}^{(1)} \ldots \exists x_{T_{i-1}}^{(1)} \exists x_{T_{i+1}}^{(1)} \ldots \exists x_{T_n}^{(1)}
  $$

  $$
  Related_R(x_{T_i}^{(1)}, \ldots, x_{T_{i-1}}^{(1)}, x_{T_i}^{(1)}, x_{T_{i+1}}^{(1)}, \ldots, x_{T_n}^{(1)})
  $$

  if $P_i$ is $full \cdot 1$ or $partial \cdot 1$, then there is a formula

  $$
  \forall x_{T_i}^{(1)} \forall x_{T_i}^{(2)} \forall x_{T_i}^{(1)} \ldots \forall x_{T_{i+1}}^{(1)} \forall x_{T_{i+1}}^{(1)} \ldots \forall x_{T_n}^{(1)}
  $$

  $$
  (Related_R(x_{T_1}^{(1)}, \ldots, x_{T_{i-1}}^{(1)}, x_{T_i}^{(1)}, x_{T_{i+1}}^{(1)}, \ldots, x_{T_n}^{(1)})) \land \\
  (Related_R(x_{T_1}^{(1)}, \ldots, x_{T_{i-1}}^{(1)}, x_{T_i}^{(2)}, x_{T_{i+1}}^{(1)}, \ldots, x_{T_n}^{(1)}) \rightarrow ((x_{T_i}^{(1)} = x_{T_i}^{(2)})))
  $$

  - there always is a formula

  $$
  \forall x_{T_1}^{(1)} \forall x_{T_2}^{(1)} \forall x_{T_3}^{(1)} ((IS \cdot A(x_{T_1}^{(1)}, x_{T_2}^{(1)}) \land IS \cdot A(x_{T_1}^{(1)}, x_{T_3}^{(1)})) \rightarrow (IS \cdot A(x_{T_1}^{(1)}, x_{T_3}^{(1)})).
  $$

  - If $E$ is a semantic constraint expression, since it is written in the same logical language, it automatically becomes a formula of the theory.
In this way we are able to transform a formal DB schema into a first order logical theory written in the many sorted logical language corresponding to the DB schema. According to the definition in logic, a theory is consistent if and only if there is no conflict implied by the theory, i.e., there exists at least one structure (model) where all the formulae of the theory are satisfied. Naturally, the definition becomes our standard to determine whether a schema is correct or not.

4.2 The meta model

In Figure 4.1 and 4.2 we present a meta model of the modeling framework in terms of the ONE-R concepts. All of the conceptual components of COMIS have been treated as entity classes and the relationships among the components are treated as relationship classes.

By using this meta model we intend to specify which components that are needed in order to form a minimally complete system specification by means of the structural constraints. In other words, the meta model shows that when a set of system specifications is considered as complete, it should at least contain some necessary components.

Since all the constraints can been seen from the meta model drawn in Figure 4.1 and 4.2, we just list some parts of it to show how the meta model works to check the completeness:

- The participation manners of the entity classes Business function and Process network diagram (PND) to the relationship class Expressed by are full.1 and partial.1 respectively; it means that a business function must be expressed by a PND, whereas a PND may express a business function (top level PND) or not (non-top level PND);

- The relationship class Flow-source-destination has three participating classes Flow, PPM_component and PPM_component, and the participation manner of Flow is full.n. This means that a flow must have a source and a destination;

- The participation manners of the entity classes Actor and Function to the relationship class Has functions are full.1 and full.n respectively; it means that an actor must have one or more functions and a function only belongs to one actor;

- The relationship class Has attribute has two participating classes Class and Data type with the participation manners full.n; this means that an entity class must have at least one data type as its attribute, whereas a data type must be used as attribute of at least one entity class;
Figure 4.1: The meta model of COMIS (1)
Figure 4.2: The meta model of COMIS (2)
4.3. Formal verification for ONE-R

- The relationship class *Belongs to basicclass* has two participating classes *Basic entityclass* and *Sub-entity class* with the participation manners *full.n* and *partial.l*; this means that any sub-entity class belongs to one and only one basic entity class, whereas a basic entity class may have no or many descendant classes;

- The remaining parts of the meta model can be explained in the same way as above.

According to the results of Sections 3.2 and 4.1, all constraints can be transformed into closed logical formulae written in a many sorted and first order language defined on the meta model. The transformations can be done in a straight way. It is not difficult to prove that the theory of the meta model is consistent since we can easily construct an example of system specification that meets all the requirements of the theory, e.g., the example used for case study in this thesis (see Chapter 6) is a correct specification corresponding to the meta model. Therefore, whenever we have built a system model with our modeling concepts, we can treat any concept as an instance of one of the classes of our meta model, and then store the instances in a specification database. By the way, when the specification database is in a consistent state according to the logical theory of the meta model, we say that the system model is complete, i.e., it contains all the necessary components required by our modeling framework.

The meta model given in Figures 4.1 and 4.2 has expressed the structural constraints by the graphical notations directly. However, if we want to make the meta model more precise, we will have to add some semantic constraints by means of explicit logical formulae.

### 4.3 Formal verification for ONE-R

According to the result of section 4.1, a formal database schema can be transformed into a first order and many sorted logical theory. In turn the consistency of the theory can be verified formally by means of logical inferences.

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

Our verification method is based on Lewis's work and Kung's work [87, 88, 83, 82], in which they defined the weakest condition to determine that if the consistency of a set of clauses is decidable, and proposed the methods to check the condition by means of graphic theory. In addition, the consistency of some
decidable clauses can be conducted directly, whereas other decidable clauses have to be checked by resolution.

### 4.3.1 Lewis's work

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

A weak condition for restricting a set $W$ of wffs is found in Lewis’s work [87]. This condition is defined in terms of the clause form of $W$. A wff is said to be in clause form if it is of the form

$$\alpha_1 \lor \alpha_2 \lor \cdots \lor \alpha_n$$

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

Definition 4.3.1: Let $S$ be a set of clauses. An $S$-link is an ordered triple $< C, \alpha, \beta >$, where $C$ is a clause and $\alpha, \beta$ are distinct literals in $C$. An $S$-chain is a sequence $< C_1, \alpha_1, \beta_1 >, \ldots, < C_n, \alpha_n, \beta_n >$ of $S$-links such that for $i = 1, \ldots, n - 1$, $\beta_i$ is unifiable with $\sim \alpha_{i+1}$; this $S$-link is said to have length $n$. If $\beta_n$ is unifiable with $\sim \alpha_1$, then this $S$-chain is called an $S$-cycle.

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

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

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

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

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

According to the definition 4.3.1, $S$ has the following 8 $S$-links, since the example has 4 clauses each of which has two literals:

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

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

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

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

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

$$< C_3, A_4(y_3), \sim A_2(x_3, y_3) >$$
4.3. Formal verification for ONE-R

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

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

\[
< C_1, \sim A_1(x_1), A_2(x_1, f_1(x_1)) > \\
< C_3, \sim A_2(x_3, y_3), A_4(y_3) > \\
< C_4, \sim A_4(y_4), \sim A_3(f_2(y_4), a_1) > \\
< C_2, A_3(f_2(x_2, a_1), \sim A_1(x_2) >
\]

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

**Definition 4.3.2:** A set \( S \) of clauses is compact if \( S \) contains no \( S \)-cycle.

**Theorem 4.3.1:** Satisfiability is decidable for compact sets of clauses.

The proof of the theorem is given in [87].

Theorem 4.3.1 means that if a set \( S \) of clauses is compact, the it can be determined by logical inference within a finite number of steps if it is consistent. Moreover, Lewis has also proved that compact sets of clauses are the maximum sets for which consistency is decidable. In other words, the above definitions determines the weakest condition for a theory to be decidable.

### 4.3.2 Kung’s work

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

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

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

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

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

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

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

**Definition 4.3.3:** A unifiable digraph \( G(S) = < S, E > \) for a set \( S \) of clauses as defined as follows.

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

**Definition 4.3.4:** Let \( G(S) \) be a unifiable digraph of \( S \). Two edges

\[
e_1 = C_{i1} \cdot \alpha_1 / C_{j1}, \beta_1 + \theta_1 \quad \text{and} \\
e_2 = C_{i2} \cdot \alpha_2 / C_{j2}, \beta_2 + \theta_2
\]

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

\[
(C_{i1} = C_{i2} \land \alpha_1 = \alpha_2) \lor (C_{j1} = C_{j2} \land \beta_1 = \beta_2)
\]

**Definition 4.3.5:** A semicycle of \( G(S) \) is said to be conflicting if it contains a node \( C \) together with a pair of its conflicting edges.

**Definition 4.3.6:** A unifiable digraph \( G(S) \) is said to be compact if \( G(S) \) contains only conflicting semicycles (if any).

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

\[
C_1: \sim WF(x_1, a) \lor WF(b, x_1) \\
C_1: \sim WF(x_2, x_2) \lor \sim WF(y_2, x_2) \\
C_3: WF(b, a)
\]

then \( G(S) \) is compact in Figure 4.3.

Based on the proof of that \( S \) is compact (Lewis's definition) if and only if \( G(S) \) is compact (Kung's definition), Kung gives the following theorem:
Figure 4.3: A compact unifiability digraph

**Theorem 4.3.2:** Satisfiability is decidable for a set $S$ of clauses if $G(S)$ is compact.

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

**Theorem 4.3.3:** Suppose that $G(S)$ is not weakly connected, and let

$< S_1, E_1 >$, $< S_2, E_2 >$, \ldots, $< S_n, E_n >$ be the weak components of $G(S)$.

Then $S$ is satisfiable iff each of $S_1, \ldots, S_n$ is satisfiable.

**Theorem 4.3.4:** If $G(S)$ contains no node of indegree zero, or no node of outdegree zero, then $S$ is satisfiable.

**Theorem 4.3.5:** If $G(S)$ is strongly connected then $S$ is satisfiable. In particular, if $G(S)$ consists of only one node, then then $S$ is satisfiable.

**Theorem 4.3.6:** If $G(S)$ consists of only a single non-conflicting semicycle, then $S$ is consistent.

A node $C$ is called a **redundant node** if $C$ contains a literal $\alpha$ which is not unifiable with the negation of any other literals in $S$. $G(S)$ is called redundant if it contains a redundant node.

**Theorem 4.3.7:** $S$ is consistent if and only if $S - \{C_i\}$ is consistent, where $C_i$ is a redundant node of $G(S)$.
4.3.3 The consistency verification for ONE-R

On the basis of Section 4.1 and Lewis’s and Kung’s work, we can define the following steps to check the consistency of a database schema:

1. collect the concepts and constraints from one or more ONE-R scenarios that are considered to be the conceptual schema of a database;
2. form the formal DB schema from the concepts and constraints;
3. transform the DB schema into a many sorted and first order logical theory in the way described in Section 4.1;
4. transform the theory into a set of clauses $S$. The method of transforming is the standard 7-step method that is given in Kung’s thesis [83] (Appendix A) and Nilsson’s books [107, 60];
5. construct the digraph $G(S)$ as defined in Section 4.3;
6. check $G(S)$ to see if it is weakly connected. If it is, then rename $S$ to $S_1$, else identify the weakly connected sets $S_1, \cdots, S_n$ each of which is maximally weakly connected.
7. For each $S_i$ ($i = 1, \cdots, n$) do the following:
   - by the theorems in Section 4.3, check if $G(S_i)$ can be determined to be consistent directly;
   - if $G(S_i)$ can not be determined to be consistent directly, then use the theorem 4.3.2 to check whether $G(S_i)$ is decidable;
   - if $G(S_i)$ is decidable, then use the resolution method to check its consistency. The method of resolution is introduced in many well known literatures such as Nilsson’s book [60]. Since $S_i$ is decidable, the resolution process will end within a finite number of steps.
8. If every $S_i$ ($i = 1, \cdots, n$) is consistent, then $S$ is proven to be consistent; if any $S_i$ is inconsistent, then $S$ is proven to be inconsistent; if any $S_i$ is undecidable, then $S$ is undecidable.

This algorithm will produce one of three possible results about the consistency of the DB schema: 1) It is consistent; 2) It is inconsistent; 3) The consistency of the schema is undecidable, i.e., can not be checked with an effective algorithm within finite steps.

If the result is the third case, it does not mean that the consistency cannot be checked. We know that all many first order theories are semi-decidable, i.e., if a theory is inconsistent, this can be checked within a finite number of steps. Therefore, even though a set $S$ of clauses is undecidable, we can still try to use the resolution method to check it. If after a large number of steps the process still can not stop, then we can remind the users about undecidability, and suggest that they try to construct an example themselves. If an example (model) can be constructed, where all the formulae of the theory are satisfiable,
then the schema is still proven to be consistent.

The proofs of all the theorems are given in [87, 83].

4.4 The static consistency check for PPM and AM

With respect to the consistency of a set of specifications for process network diagrams and actors, we divided the work into two parts. We first check the structural aspects, i.e., the aspects that are irrelevant to the dynamic executions of the PNDs. The static check is not enough, so we must also use the dynamic method to be introduced in the next section.

First, we can use the logical method to check some properties. Unlike for the formal verification of a DB schema, we only give following rules for the structural consistency of ports, the consistency of i/o conditions of processes, the structural consistency of the PNDs, and the structural consistency for process decompositions.

The rules are:

S1. Every process has at least one CIP and at least one COP;

S2. A CIP has at least one triggering input, and a COP has at least one terminating output;

S3. The i/o condition of a process must ensure that every CIP satisfies at least one COP, meanwhile any COP must be satisfied by at least one CIP. We say that a CIP satisfies a COP if and only if

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

S4. The i/o condition of a process must not include a conjunction term of inputs that can not be in any CIP of the process;

S5. The set of the members of a canonical port can not be a subset of another canonical port;

S6. Only a singular input/output can be a triggering/terminating input/output;

S7. If in the i/o condition of a process all conjunction terms of an output contains at least one conditional input, then the output must also be a conditional output;

S8. If a group of inputs are the triggering inputs of a CIP for the higher level
processes, then in the process network that is the result of a decomposition of
the process, they must also be the triggering inputs of a CIP for one sub-process
in the network; if a group of outputs are the terminating outputs of a COP for
the processes, then in the process network, they must also be the terminating
outputs of a COP for one sub-process in the network.

If we express the rules in terms of logical theory, then it is a consistent theory
since a model of the theory is found in the example of Figure 3.23 which is
transformed into the canonical form of Figure 3.29. Therefore, we can just use
these rules to check the structural properties of processes, functions of actors,
i/o conditions and PNDs.

When a PND is the result of the decomposition of a process, an important
issue is to ensure that the PND has the property of constructivity, i.e., if it
uses the same inputs and produce the same outputs as the process. This topic
must be discussed in the context of the dynamic executions of the processes in
the PND. This will be dealt with in the next section.

However, the i/o conditions of the PND can be calculated without considering
the dynamic executions. For example, Tao and Kung proposed to calculate the
transitive closure of the precedence relation of a DFD [144] in order to check the
correctness of a decomposition with respect to the precedence relations between
the inputs and outputs of a decomposed process. In PPM, differently, we
use i/o conditions that provide more accurate information about the relations
between an output and some of the inputs of a process. We thus use the
following substitution algorithm to calculate the i/o condition expression of an
output in the PND which is the decomposition of a process, with the similar
principle to that of Tao and Kung:

Algorithm Calculation-of-Input-Output-Condition
input: An output O of the process network;
output: The i/o condition expression for O on the external inputs of the network;

begin
in the i/o conditions for the subprocesses which has output O
{ O and inputs to the subprocess are given different numbers in the
sub-processes respectively }
collect all n rows for O, form the expression
\[ S_0 = T_1 \lor \cdots \lor T_n, T_i = I_i \land \cdots \land I_i \]
{ \( i \geq 1 \) and \( I_i \) is an input for a subprocess } and let \( S_0 \) be the current expression;
repeat
if in the i/o condition of a sub-process which has \( n_1 \) inputs and outputs O
there is no row for O
then begin form the expression \( S_1 = I_1 \lor \cdots \lor I_{n_1} \);
{ \( I_j \) is the name of an input to the sub-process, \( 1 \leq j \leq n_1 \) }
{ just assume there are \( n_1 \) rows for O, each of which shows an input is used }
let \( S_{O_1} \) be the disjunction of the current expression and \( S_1 \);
let $S_{O_1}$ be the current expression;
until all such sub-processes have been found;
repeat do on the current expression $S$
begin
if any input $I$ is linked to the output(s) $O_1, \ldots, O_m (m \geq 1)$
of the subprocesses in the process network through some flow-pipe(s)
then substitute $I$ with $C_1 \lor \cdots \lor C_m$
{ $C_k (1 \leq k \leq m)$ is the condition expression for $O_k$ }
and let the new expression after the substitution be the current expression;
if $I$ is linked with a data store and the data store receives data from
outputs $O_1, \ldots, O_v$
then substitute $I$ with $C_1 \lor \cdots \lor C_v$
{ $C_k (1 \leq k \leq v)$ is the condition expression for $O_k$ }
and let the new expression after the substitution be the current expression;
if $I$ is linked with a data store and the data store does not receive
any data within the process network
then substitute $I$ with $I_1 \lor \cdots \lor I_g$
{ $I_1, \ldots, I_g$ are all the external inputs to the process network }
and let the new expression after the substitution be the current expression;
transform the current expression into a disjunction normal form $S_n$;
delete from $S_n$ all the conjunction terms whose inputs are external inputs to
the network but they can not appear together in any CIP of the higher
level process, and let the left parts of $S_n$ be the current expression
end
until the content of $S$ can not be changed by the above substitutions or $S$
has been the same with one of the old expressions:
delete all the conjunction terms that contain an internal input from the
current expression;
let the left part of the current expression be the condition expression of $O$ for the
process network;
end (algorithm)

This algorithm is used for each output of a process network to conduct its i/o
condition expression on the external inputs to the network. First the condition
expression(s) for the output of the sub-process(es) is taken or merged into a
“current expression”, then the input names in the expressions are substituted
with the condition expressions for the outputs of other processes that are linked
with the inputs through flow-pipes in the process network, while those parts
in the expression with illegal combination of external inputs are deleted. The
substitution is done recursively until the current expression can not be changed
any more. After deleting the terms containing internal outputs produced in
some loops of the network, the final expression is just the i/o condition expreession
of the output on the process network. If the result is empty, then the
process network is inconsistent because an output can not get the data needed
from the outside of the network during an execution of the process network.
The result should also be compared with the i/o condition of the decomposed
process.

In Figure 4.4, we calculate the condition expression for the outputs $o_1$ and $o_2$
The condition expression for P1: o1
(P1.4:o1)
\[(P1.4:11 \land P1.4:13) \lor (P1.4:12 \land P1.4:13)\]
\[\downarrow\]
\[\downarrow\]
(V (P1.2: 11 \land ( (P1.3:11 \land P1.3:13) \lor (P1.3:12 \land P1.3:13)) \lor P1.2:11)
\[\downarrow\]
\[(P1.3:11 \land P1.3:12 \land P1.3:13)
\lor (P1.3:11 \land P1.3:12)
\lor (P1.3:11 \land P1.3:13)
\lor P1.2:11\]
\[\downarrow\]
\[(P1.1:11 \land P1.3:12 \land P1.3:13)
\lor (P1.1:11 \land P1.3:12)
\lor (P1.1:11 \land P1.3:13)
\lor P1.2:11\]
\[\downarrow\]
\[(P1.3:11 \land P1.3:13) \lor P1.2:11\]
\[= (P1:11 \land P1:12) \lor P1:13\]

The condition for P1:o2 (P1.4:o2) is the same as the above expression

Figure 4.4: The i/o condition for the process network for P1 of the IFIP ticket booking activities
of $P_1$ in the process network shown in Figure 3.23 or Figure 3.29 (ports are transformed into the canonical forms). The result is same as the original i/o condition of $P_1$, thus we can conclude that the decomposition is correct on the aspect of i/o conditions.

4.5 The dynamic consistency check for PPM

In this section, we present the method for dynamic consistency check for PPM. More specifically, it is a method to check the constructivity property of the decompositions by calculating the possible execution sequences of a process network.

The reason that we use the dynamic method comes from the conclusion that the constructivity property is difficult to be checked in a static way. In sequel, we will first introduce other relevant work on this issue and conduct the conclusion, and then introduce the algorithms of our STD method.

4.5.1 Some possible ways for checking the constructivity

Until now, little work has been done on the issue of constructivity except the work of Sindre [135] and Kung [85]. In Sindre's report [135], he claimed that he would concentrate on the problem of "constructivity", i.e., derive the properties of a system from the relevant properties of the system's components, and use this method to compare the properties of a process and the process network produced from a decomposition of the process. Kung also wrote in his paper [85] "the network of subprocesses resulting from decomposition must be consistent with the higher-level process", and treated this as the goal of his method for consistency checking. Since they did their works in different ways, we will take a short investigation on them and discuss on them.

In Kung's approach [85], the inputs and outputs to a process are treated as logical expressions with three operators conjunction ($\cdot$), disjunction ($\odot$) and exclusive disjunction ($\oplus$) showing the logical combination of the data flows. To check the decomposition of a process, a logical inference is taken with the inputs as a given condition. The rules used during the inference are of two types: logical rules and non-logical rules, i.e., each sub-process is regarded as a rule by which we infer some data flows (outputs of the sub-process) from some existing data flows (the inputs to it). For instance, from the $P_{1.1}$ in Figure 4.5, we get a rule $P_{1.1} \models a \Rightarrow f_{1.1} \oplus f_{1.2}$. The final deducted expression can be used to compare with the output expression of the higher level process.

Sindre [135] constructs a state transition diagram on the process network which
is the result of the decomposition of a process. He defines the state vectors making up a state space, and the event space consisting of the events of type $r_f$ (receive a datum from flow $f$), $s_f$ (send a datum to flow $s_f$), $br_f$ (start to receive data from repeating flow $f$), $er_f$ (end receiving data from repeating flow $f$), $bs_f$ (start to send data to repeating flow $f$) and $es_f$ (end sending data to repeating flow $f$). The construction starts at an initial state, then branches are formed by choosing an event from the possible event set at the current state (the event will result in a new state). The same procedure is then repeated for all the new states until every branch has reached the STOP state (all the termination flows have been sent out). The produced graph is used by an algorithm to conduct the port structure for the outputs of the network: the sequences are used to construct and ports, the branches are used to construct xor ports, and the loops are used to specify repeat and conditional flows. The constructed port structures will be simplified with some equivalence rules, and are then used to compare with the port structures of the higher level process.

![Figure 4.5: A decomposition of a process $P_1$](image)

In Figure 4.6, we show the two ways for consistency check on the decomposition given in Figure 4.5.

With respect to Kung’s approach, a fundamental weak point is that the pure logical method uses static means to deal with the inputs and outputs of the dynamically and concurrently executing processes. Consequently, whether a set of input flows and output flows contain data, i.e., if the logical expressions about the flows are true or false, changes over time, whereas in an ordinary logical system a true statement always keeps its truth in the whole inference process. Therefore, wrong conclusions might be drawn when we use this method. This problem has been revealed by the examples given in a paper [27] on the same issue and in the same journal where Kung’s paper was published, which
4.5. The dynamic consistency check for PPM

Nonlogical rules:

\[ P_1.1 \models a \Rightarrow f_{1.1} \oplus f_{1.2} \]
\[ P_1.2 \models f_{1.1} \oplus b \Rightarrow f_{1.3} \]
\[ P_1.3 \models f_{1.2} \oplus c \Rightarrow f_{1.4} \]
\[ P_1.4 \models f_{1.3} \oplus f_{1.4} \Rightarrow d \oplus e \]

<table>
<thead>
<tr>
<th>step #</th>
<th>results inferred by the step</th>
<th>rules used at the step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>(a \bullet b \bullet c)</td>
<td>given input</td>
</tr>
<tr>
<td>2)</td>
<td>(a)</td>
<td>logical rules, 1)</td>
</tr>
<tr>
<td>3)</td>
<td>(b)</td>
<td>logical rules, 1)</td>
</tr>
<tr>
<td>4)</td>
<td>(c)</td>
<td>logical rules, 1)</td>
</tr>
<tr>
<td>5)</td>
<td>(f_{1.1} \oplus f_{1.2})</td>
<td>logical rules, P1.1, 2)</td>
</tr>
<tr>
<td>6)</td>
<td>(f_{1.3} \oplus f_{1.4})</td>
<td>logical rules, P1.2, P1.3, 3, 4, 5)</td>
</tr>
<tr>
<td>7)</td>
<td>(d \oplus e)</td>
<td>logical rules, P1.4, 6)</td>
</tr>
</tbody>
</table>

a) Kung’s consistency check by logical inference

b) Sindre’s State Transition Diagram

Figure 4.6: Two ways for consistency check on the decomposition of \(P_1\)
provides some comments on Kung’s method.

There is another big unsolved problem in both Kung’s and Sindre’s works: is a decomposition itself feasible? In other words, can the network of processes execute until some outputs are sent to the outside world? If the processes combine in an erroneous manner, then it is meaningless to talk about the correctness of the outputs from the network!

![Diagram](image)

**Figure 4.7:** Two process networks which may produce run-time errors

In Figure 4.7 we give two examples to show how errors may be generated from inappropriate network structures. In Figure 4.7.a, process $P$ terminates when its and output port sends two data items into flow $f_2$ and $f_3$ simultaneously. Now process $Q$ should be triggered. However, the xor input port of $Q$ requires that one and only one input flow is full, so the output of $P$ is actually an illegal input to $Q$. Figure 4.7.b gives another version of decomposition of process $P_1$ in Figure 4.5. $P_{1.1}$ now has an and output port, so that the flow $f_{1.1}$ and $f_{1.2}$ are full simultaneously when $P_{1.1}$ terminates. It may give at least two results: if $P_{1.2}$ and $P_{1.3}$ was exactly same length of time, then $P_{1.4}$ will get an illegal input; otherwise $P_{1.4}$ will possibly be triggered two times. That is still not what we expect.

This problem is not mentioned in Kung’s paper. Because errors often happen during the dynamic executions of the processes, it is definitely difficult to use Kung’s method to detect the flaws of a process network structure. In the other approach, Sindre had noticed the problem and discussed on it. His solution is the definition of a standard to determine if a network of processes is a legal decomposition of some higher level process:

1. *All the processes in the network satisfy the PPM syntax;*
2. There is one unique process which is triggered first in every possible execution of the network;

3. There is one unique process which terminates last in every possible execution of the network.

The last two rules are used to prevent the so-called "mingling" problem, i.e., the problem raised by the subsequent executions of a process network and the concurrent executions of the processes within the network – the data items in different executions might be wrongly received and processed. A similar problem is also revealed in [27].

These two rules seem to be too strong to be accepted in a practical methodology, e.g., in the ESPRIT project TEMPORA the rules are not followed [13]. Moreover, even when we have imposed the constraints, mingling problem may still happen in concurrent processes between the first and last processes. Therefore, we would rather give up these two rules and leave the problem open. In this thesis, we only consider the problems of the separated executions of the higher level process when it is decomposed into a PND.

Now we turn to the first rule given by Sindre. It states that when all the processes in a network satisfy the PPM syntax, then the network is a legal one. Just think of the examples in Figure 4.7 where all the processes have been drawn correctly, but where errors can still happen. Thus, our conclusion is: When we decompose a process into a network of processes, if not only each process in the network satisfies the PPM syntax, but also the interrelationships among them will not produce execution errors, then we can say that the decomposition is legal or safe.

Such properties, as analyzed above, are difficult to check in a static way. Therefore, we choose State Transition Diagram (STD) as the basis of our method. First of all, we define the standard of consistency of a decomposition as:

1. The network can execute smoothly until the final outputs to the outside of the network have been produced;

2. The network receives the same inputs and produces the same outputs as the decomposed process does.

4.5.2 Construction of an STD through Canonical Process Ports

In the previous section, it has been pointed out that the more effective way of analyzing a process network is to construct an STD. However, because of the randomness of the events, even the STD for a PND with only a few processes
might be a mess. To be able to construct an STD that is possible to analyze, we must find out a way for simplifying. The canonical port structure just meets such a requirement.

![Diagram of process P with port structure](image)

**Figure 4.8:** Conducting possible execution sequences of a process

Looking at the example given in Figure 4.8. When we consider the execution of process $P$ given in Figure 4.8.a, since it is a composite port, we must check if both the sub-port $\text{xor}(a, b)$ and flow $c$ are filled, then by the definition of $\text{xor}$ port, we must also check if flow $a$ or $b$ is filled. The similar work must also be done at the output side. Moreover, if the ports are more complex, then the possible executions will be so many that the STD of it will be difficult to be constructed.

However, when the ports are transformed into the form in Figure 4.8.b, the possible executions may be know immediately for that we know that during any execution ONLY ONE CIP and ONLY ONE COP are used. Even if the original port structures are more complex, the property of canonical ports keeps unchanged, since the structure is unique. Moreover, we know that the canonical port structure will not affect the meaning of the process since we are only interested in information about the inputs and outputs in an execution of the process.

This is shown in Figure 4.8.c. Assume that any CIP enables all COPs during different executions, if the CIP and $(a, c)$ is used during the execution, then we immediately know two possible executions may happen at next time, with the
COP and$(e, g)$ and and$(f, g)$ used respectively.

Another advantage is in analyzing the behavior of the process during the process. Since CIP and COP are all and ports, we know that all inputs of the CIP will be received as well as that all outputs of the COP will be sent out. Therefore, the behaviors of all processes during any execution can be assumed as the sequence given in Figure 4.9. The calculation is thus simplified both by the canonical port structure and by the assumption about the behavior of a process during an execution.

Now we define the data structure for states and then make some assumptions about the possible executions before further design the algorithms for the calculations.

The data structure for state vector

There is a lot of information associated with the dynamic executions of a process network. However, because the specification of a process network in general provide only incomplete knowledge about the process, we can only obtain limited data which reflect on the execution states of the network. We are interested in three kinds of states: 1) a flag showing whether a network as a whole is in a normal state; 2) the state of the processes in the network; 3) the states of the flow-pipes in the network. At any time, the state of the network is comprised of the states of the network component, and represented by a state vector. Now we introduce the data structure of the state vector in a PASCAL-like syntax:

```pascal
type state_vector =
record
  normal_system_state : boolean;
  {there are N processes in the network}
  {there are M flow-pipes in the network}
end;

type state_of_process =
record
  running: boolean;
  CIP: CIP_state; {one CIP is used during an execution of the process }
  COP: COP_state; {one COP is used during an execution of the process }
end;

type state_of_flow_pipe =
record
  volume: integer; {the maximum capacity of the flow-pipe}
  has_data: integer; {the sum of data staying in the flow-pipe at a moment}
end;
```
type CIP_state =
  record
    ID: integer; {the identifier of the CIP in the process}
    receive_events: array[1:NI] of event; {NI possible kinds of events at the CIP}
  end;

type COP_state =
  record
    ID: integer; {the identifier of the COP in the process}
    send_events: array[1:NO] of event; {NO possible kinds of events at the COP}
  end;

type event =
  record
    event_name: string;
    happening_times: integer; {the times that the event has happened during
      the current execution of the process}
  end;

The semantics of the data structure is explained as follows:

- \textit{normal-system-state} indicates whether the network is in a legal state;

- if the state variable \textit{running} in a process state is true, then the process is running. Because at any execution of a process, one and only one CIP as well as one COP, is used, we will also keep information of the two active canonical ports;

- for a CIP, we need to know its ID, and how many times the event of receiving any of the inputs in the CIP has happened (for a singular input the event will happen only once, but for a repeating input the event may happen more times). Similar information should also be kept about the COP.

- For any flow-pipe, we keep a record about its volume (for a flow-pipe only lined to singular output the volume is 1, whereas for a flow-pipe lined to repeating output the volume is a integer $M > 1$), and how many data items have been put into the flow-pipe.

The assumptions on the behavior of a process network
Because we can not know the execution details of the processes, we have to make some assumptions about the behavior of a process network. Two factors are taken into account: 1) the assumptions should be as close as possible to the general behavior of process networks; 2) the assumption should depend only on the information that we can obtain from the specification of the network.

Now we list the assumptions:
1. only the events of receiving data and sending data are considered;

2. during an execution of a process, one and only one CIP, as well as one and only COP, is to be used;

3. during an execution of a process, a singular input/output will be received or sent once, whereas the number of receiving or sending of a repeating input or output may vary. At the input side, a repeating input will be received at least once from each flow pipes linked to it and receive from every flow-pipe all the available data items. At the output side, if a repeating output depends on a singular input by the i/o condition of the process, it will be sent out twice; if it depends on a relating input, however, it will be sent out as many data items as that the the repeating input receives.

4. the outputs of a process depend only on its inputs. This means that whenever an output condition is satisfied, it will send the output to the corresponding flow-pipe without considering whether the flow-pipe is full or not;

5. the inputs from outside of the boundary of the network can always be received;

6. a flow-pipe linked to a data store as a data resource can always provide a datum (or data) to the corresponding input or receive a datum (or data) from the corresponding output;

7. the life cycle of an execution of a process is

   begin {triggered by arrivals of a group triggering inputs to the CIP}
      receive all the triggering inputs;
      while the process has not received all non-conditional inputs and sent out all the non-conditional and non-terminating outputs do
         begin
            receive all the inputs that have been put into the flow-pipes linked to the inputs;
            if nothing is received (data have not arrived or all input actions have ended)
               then send out all available non-terminating outputs;
         end {while}
      send all the terminating outputs of the COP;
   end; {the execution}

8. a receiving or sending action costs very little time, so during concurrent executions of the processes, we can always consider a group of receiving or sending actions of a process at a particular time point as an execution unit which is atomic relative to the concurrent executions of the processes. In Figure 4.9, we show these “packaged events” during the execution cycle of a process.

9. whenever a flow-pipe gets more data than its capacity, or triggering inputs arrives to a running process, or two different sets of triggering inputs which can trigger two CIPs of a process are full simultaneously, the pro-
cess network has fallen into an error state and can not go on with its execution.

![Execution Life Cycle Diagram](image)

Figure 4.9: The execution life cycle and the operation groups of a process

The algorithms to construct an STD

We use a directed graph to represent an STD: each node represents one or more state vectors and each edge is marked with one or more events that change the state of the process network at one time point to another state at the next time point. When a node represents two or more state vectors, then the possible consequent events for each of the state vectors will be exactly same. Any node represent only one state vector when it is created, but may be attached more vectors during the construction of an STD. The time interval between the states is not fixed, because we assume that each process can perform receiving or sending actions in a short time, and that it can carry out all the possible operations subsequently and independently without being interrupted by other processes. Therefore, what causes the change of one state into another is not a "single event", but a set of events occurring in a process.

Following is the algorithms for the state-transition. The first one is to check out all possible event sets at a specific state.

```plaintext
type set_of_event_sets = set of events;

type events = set of string; // a set of event names

Algorithm Find-Possible-Event-Sets(S : state_vector,
    var PESS : set_of_event_sets)

parameter: input: a state-vector S; output: the set of all possible event sets at the state;

begin
    PESS := Ø; // it is set to empty initially
    // search all possible event sets
    for each process P in the network do
        begin
            if for the process state of P in S
                S.states_of_processes[i].running = false
                { i is the index of the process state for P in S }
```
4.5. The dynamic consistency check for PPM

then \( P \) is idle

begin

if for every triggering input of a CIP of \( P \)
\( S.\text{states.of_flow_pipes}[j_0].\text{has_data} \neq 0 \)
\{ \( j_0 \) is the index of the state of the flow-pipe linked to
the member of the triggering input group in \( S \) \}
then insert \{receive\_i_{j}, \ldots, receive\_i_{j}\} into PESS;
\{that \( P \) is triggered and then receives the triggering inputs is a
possible event\}

end
else \( P \) is running

begin

{check if \( P \) can terminate}

if for any non-triggering and non-conditional input in the CIP and any
non-terminate and non-conditional output in the COP,
\( S.\text{states.of_processes}[i].\text{CIP.receive_events}[k_1].\text{happening_times} = e_1 \)
\{ \( e_1 \) is 1 for a singular input, \( \geq 2 \) for a repeating input\},
\{ \( k_1 \) is the index of the input in \( S.\text{states.of_processes}[i] \) \}
\( S.\text{states.of_processes}[i].\text{COP.send_events}[k_2].\text{happening_times} = e_2 \)
\{ \( e_2 \) is 1 for a singular output, \( \geq 2 \) for a repeating output\},
\{ \( k_2 \) is the index of the output in \( S.\text{states.of_processes}[i] \) \}
and \( e_1, e_2 \) satisfy the condition specified in the assumptions,
then \( P \) can terminate

insert \{send\_o_{m_1}, \ldots, send\_o_{m_i}\} into PESS;
\{o_{m_1}, \ldots, o_{m_i} \) are all the terminating outputs of the COP of \( P \)\}
else \( P \) cannot terminate

begin

if for a flow-pipe \( f \) which is linked with an input \( I \) in the CIP
\( S.\text{states.of_flow_pipes}[h].\text{has_data} \neq 0 \) \{the linked flow-pipe has data\}
\{ \( h \) is the index for the state of the flow-pipe is \( S \) \}
and \( S.\text{states.of_processes}[i].\text{CIP.receive_events}[c].\text{happening_times} < e_1 \)
\{there is at least one input to be received \}
then
find out all such inputs \( i_{i_1}, \ldots, i_{i_m} \) and insert
\{receive\_i_{i_1}, \ldots, receive\_i_{i_m}\} into PESS
else \{no input to be received\}

begin

if for a non-terminating output \( O \) in the COP
\( S.\text{states.of_processes}[i].\text{COP.send_events}[c].\text{happening_times} = e_2, \)
\{ \( e_2 \) does not satisfy the condition specified in the assumptions
and the output is ready according to the i/o condition of \( P \)
and the state of CIP of \( P \)
\{there is at least one non-terminating output to be sent out\}
then find out all such outputs \( o_{g_1}, \ldots, o_{g_n} \) and insert
\{send\_o_{g_1}, \ldots, send\_o_{g_n}\} into PESS
end

end

end

end \{ all possible event sets for process \( P \) have been found\}
{all the possible events sets for the whole process network have been found\}
end (the algorithm)

The second algorithm builds new state nodes from a state node $S$ and calculate all the new states resulting from the corresponding event groups. Every new state is checked to see if it is a consistent state. If it is, then the algorithm is called recursively to calculate more possible states; otherwise it is marked as an inconsistent state.

**Algorithm** State-Transition($S : state\_vector, PESS : set\_of\_event\_sets$)

**parameter:** a state-vector $S$ and the possible event sets $PESS$ at the state;

**result:** extension of the state transition diagram with possibly more nodes from the state $S$;

begin

```
var NEW_PESS: set\_of\_event\_sets;
if $PESS = \emptyset$ {No events may happen at state $S$}
themark the node for $S$ as a STOP node
else {there are possible event sets in $PESS$}
begin

for each event set $ES$ in $PESS$ do

begin

```
case $ES = \{receive_{ij_1}, \cdots, receive_{ij_l}\}$
and the $l$ inputs are triggering inputs to the process $P$:
begin

calculate all possible $m$ CIP/COP pairs on the triggering condition
{ a triggering group inputs may trigger more than one CIPs, and
 each CIP may produce outputs for more than one COP }
create $m$ new nodes with state vectors $S_{n_1}, \cdots, S_{n_m}$ and with the
CIP/COP pairs respectively and $m$ edges from the node for $S$ to them,
all marked with the
value of $ES$ receive$_{ij_1}, \cdots, receive_{ij_l}$;
assign value to the state_vectors to express that:
process $P$ is now running;
the flow-pipes for the triggering inputs are now empty;
{ the has_data items in $S_{n_k}$ for the flow-pipes is 0 now }
The events receive$_{ij_1}, \cdots, receive_{ij_l}$ have happened once;
All other events for $P$ have not happened yet;
end
```

```
case $ES = \{send_{o_{j_1}}, \cdots, send_{o_{j_s}}\}$
and the $s$ outputs are terminating outputs to the process $P$:
begin

create a new node with state_vector $S_n$ and an edge from the node
for $S$ to it;
mark the edge with the value of $ES$;
assign value to the state_vector $S_n$:
first copy the value of $S$ to $S_n$;
then change the data so that $P$ is idle and all other data for $P$ is cleaned
and each of the flow-pipes linked to the outputs has got a new datum ;
{ the has_data items for in $S$ is added with 1 if the flow-pipe is not
```
4.5. The dynamic consistency check for PPM

linked with a data store
end

\text{case } ES = \{\text{receive}_{ij_1}, \ldots, \text{receive}_{ij_l}\}
\text{and the } l \text{ inputs are non-triggering inputs to the process } P:
\text{begin}
\begin{itemize}
  \item create a new node with state_vector } S_n \text{ and an edge from the node for } S \text{ to it;}
  \item mark the edge with the value of } ES;\)
  \item assign value to the state_vector } S_n:\n    \begin{itemize}
      \item first copy the value of } S \text{ to } S_n;
      \item then change the data so that the } has_data \text{ items and } happening\_times \text{ in } S \text{ for the inputs and the corresponding flow-pipes changed;}
    \end{itemize}
    \begin{itemize}
      \item for any receive operation, if the input is not an external and repeating input and the flow-pipe is not linked with a data store,
      \item then the } has\_data \text{ of the flow-pipe is reduced with } 1; \text{ The } happening\_times \text{ for the receive event is added with } 1 \}
\end{itemize}
end

\text{case } ES = \{\text{send}_{o_j}, \ldots, \text{send}_{o_j}\}
\text{and the } s \text{ outputs are non-terminating outputs to the process } P:
\text{begin}
\begin{itemize}
  \item create a new node with state_vector } S_n \text{ and an edge from the node for } S \text{ to it;}
  \item mark the edge with the value of } ES;\)
  \item assign value to the state_vector } S_n:\n    \begin{itemize}
      \item first copy the value of } S \text{ to } S_n;
      \item then change the data so that the } has_data \text{ items and } happening\_times \text{ in } S \text{ for the inputs and the corresponding flow-pipes changed}
    \end{itemize}
    \begin{itemize}
      \item for any send operation the flow-pipe has one more datum if it is not linked with a data store, and the send event is added with } 1 \}
\end{itemize}
end;
\{ \text{all the new nodes have been created} \}
\{ \text{now check the consistency of every new node } \}
\text{for each new node with its state_vector } S_n \text{ do}
\begin{itemize}
  \item if } S_n \text{ falls into one of following cases:
    \begin{itemize}
      \item two CIP with different triggering inputs of an idle process may be triggered;
      \item a CIP of an executing process may be triggered;
      \item a flow-pipe contains the data more than its volume;
    \end{itemize}
  \item then } S_n.\text{normal.system.state} := \text{false}; \{ \text{mark } S_n \text{ as "inconsistent state"} \}
\end{itemize}
\{ \text{now check any possible loop structure in the STD} \}
\text{for each new consistent node with its state_vector } S_n \text{ do}
\begin{itemize}
  \item if } S_n \text{ is same with a state_vector } S_o \text{ of a node which has exited in the STD being constructed, or it is same with } S_o \text{ in all other part but that for a process } P \text{ and all the flow-pipes linked with the inputs of } P, \text{ and the events are triggering } P \text{ at a different CIP with that of } S_o \text{ but all other consequent events for } P \text{ will be same with that in } S_o
  \item then begin \{ \text{a loop may have been found} \}
    \begin{itemize}
      \item erase the node and the edge from the node for } S \text{ to it;}
      \item attach } S \text{ to the node for } S_o;
      \item create an edge from the node of } S \text{ to that of } S_o, \text{ and marked the edge with the event set used for marking the removed edge;}
    \end{itemize}
  \end{itemize}
\text{end}
\{ \text{recursively call the algorithm to expand the STD} \}
for each new consistent node with state_vector $S_n$ do
begin
    call Find_Possible_Event_Sets($S_n$, NEW_PESS);
    call State_Transition($S_n$, NEW_PESS);
end
end (algorithm)

On the basis of the two algorithms, we now illustrate the algorithm to construct the STD of a process network which is a decomposition of a higher level process $P$. We will first build a START node representing the initial state $S$ when the network is idle, then collect all the triggering event for $P$ as the possible event sets, and call the previous algorithm to construct the STD from $S$.

**Algorithm Construct-STD**

input: The information for a process $P$ and a process network $N$ as the decomposition of $P$, including the canonical port structures and i/o conditions for $P$ and all the processes in $N$;

output: An STD which shows the possible execution cases of $N$;

begin
    var PESS: set of event_sets;
    create a START node with an initial state $S$ at which all processes in $N$ are idle and all flow-pipe are empty;
    $S.normal.system.state := true; \{ \text{mark the state as "consistent state"} \}$
    create $n$ new nodes and $n$ edges from the START node to them;
    \{ The higher level process $P$ has $n$ CIPs \}
    mark any edge with the event "arrival of data at the inputs" on a particular CIP \{ the special events only be recorded at the initial state \}
    for each new node do
begin
    create a new state_vector $S_n$ and copy the value of $S$ to $S_n$;
    update the value of $S_n$ so that the flow-pipes for a CIP of $P$ is full;
    \{ every corresponding flow-pipe has a datum \}
    if there is any conditional input in the CIP
    then begin
    create a new node and an edge from the START node to it with the same events;
    copy the state_vector $S_n$ to $S_{n1}$ as the state_vector of the new node;
    update $S_{n1}$ so that all the flow-pipes linked with the conditional inputs are empty;
end;
\{ create another state_vector where the conditional inputs are empty \}
end (for)
for each new node with a state_vector $S_{new}$ do
begin
    call Find_Possible_Event_Sets($S_{new}$, PESS);
    call State_Transition($S_{new}$, PESS);
end (for)
end (algorithm)

Figure 4.10: The STD of the process network for $P_1$ in the IFIP ticket booking activities

The algorithms are written in a semi-natural language (English + PASCAL style) for easier understanding. In Figure 4.10, we draw the STD for the process network in Figure 3.29 in order to show an example. A small circle represents the state vector attached to a node in the graph. Usually only one state vector is put on a node, but sometimes a node may have more state vectors. In Figure 4.10, the node B has two state vectors showing that the process $P_{1.3}$ (see Figure 3.29) may be triggered with a datum at its input $i_1$ or $i_2$. In the first case, the process $P_{1.1}$ just terminates and sends a datum into the flow-pipe $f_{1.1}$; in the second case, the process $P_{1.2}$ just terminates, instead, and sends a datum into the flow-pipe $f_{1.2}$. Both states will result in exactly the same subsequent events, either using the canonical port for $o_1$ or that for $o_2$ and $o_3$, so they are attached to the same node. Node A and B also show another feature of the STD: a state can be changed into different new states by the same event (for node A it is the event receive$_{i_1}$). This shows that when a process is triggered in one CIP, it may have different CIP/COP pairs in its executions (also see process $P_{1.3}$ in Figure 3.29). In the STD, every branch from the START node can reach a STOP node, so we say the process network is feasible.

Another example is given in Figure 4.11, where the canonical port structure and a part of the STD of the process network in figure 4.7.b) are shown. The i/o
Figure 4.11: The canonical port structure and the STD of the process network
conditions are omitted since no non-terminating outputs exists in any process (thus we assume one terminating output can always be sent when the process is triggered). Node A is inconsistent, because when process P1,4 is still running, process P1,3 sends a datum from its output o1, then the flow-pipe f1,4, which can trigger P1,4 from its input i2, is full. This leads to an inconsistent state. Node B is also inconsistent, because process P1,4 is triggered and sends data into the flow-pipe e twice, and the capacity of the singular flow-pipe e is only 1. In fact, all paths from node START will lead to an inconsistent state, so we say the process network is infeasible.

Figure 4.12: The canonical port structure and the partially feasible STD of a process network

The above two examples show the extremities of executions. As a matter of fact, in analyzing the various process networks, most of them will behave differently along the different paths: some of the paths will lead to consistent STOP states and some others will lead to INCONSISTENT states. An example is given in Figure 4.12. We say such an STD is partially feasible.
Because the construction of an STD is only to simulate the execution of a process network approximately and the details for control of the execution is ignored, it is possible that a process network is practically correct even though its STD is only partially feasible. However, if a network is totally infeasible, we then have enough reason to say it is not a correct structure and reject it.

4.5.3 Synthesis for the Properties of the Process Network

![Diagram of an STD and all its paths]

Figure 4.13: An STD and all its paths

When a process network is feasible or partially feasible, a further question about the network is if it has the same properties as the higher level process which is decomposed into the process network, i.e., if the network produce the same outputs as the decomposed process? Meanwhile, if the outputs to the outside of the network have the same relationships with the inputs from the outside as those specified in the i/o condition of the process?

Of the two problems, the second one has been solved by static check. The algorithm has been presented in Section 4.4.

To solve the first problem, we can construct the port structure for the network. We first find all the possible execution paths from the START state to a consistent STOP state. The example in Figure 4.13 shows the principle to identify paths: each path with a particular node set is selected and identified, and the nodes within a loop should appear in a set only once except the node at which the loop begins and ends.

Each path shows an execution of the process network, thus the receive events at the external inputs imply an and input port containing these inputs. Similarly, the send events at the external outputs imply an and output port. Because in an execution the process can only follow one path, it is also implied that the and ports on the different paths will compose an xor port. However, it must be recognized that two different paths may have the events for receiving or sending data at the same or similar inputs or outputs, so the and ports for them should first be merged. On the other hand, the fact that some event may appear more than once or be within a cycle implies that some inputs or outputs are repeating and/or conditional. All the considerations have been represented in the following algorithm:
Algorithm Construct-Input-Port-For-Process-Network
input: An STD for a process network and all other data for the network;
output: An input port structure for the process network;

begin
    identify all $n$ possible paths;
    for each path $Path_i$ ($1 \leq i \leq n$) do
        create a list which includes all the receive events for the inputs from external
        flow-pipes along the path;
        from all the list $List_1, \ldots, List_n$, select out $List_{c1}, \ldots, List_{cm}$ that the event
        set in each of them is not proper subset of the event set of any other list among
        $List_1, \ldots, List_n$ and any lists with exactly same members are merged into
        a single list;
        for each list among $List_{c1}, \ldots, List_{cm}$ do
            create an and port $P_i$ ($1 \leq i \leq m$) whose members are the inputs appear
            in the list;
            create an xor port $PI = xor(P_1, \ldots, P_m)$;
        for each input $I$ from external flow-pipe do
            begin
                if the event for receiving $I$ does not appear in a list and the event set of the list
                is a subset of the event set of any list among $List_{c1}, \ldots, List_{cm}$
                then mark $I$ as conditional input;
                if the event for receiving $I$ appears in any list and it appears twice or more
                in a list among $List_{c1}, \ldots, List_{cm}$ and at least it appears
                once outside any loop in the paths
                then mark $I$ as a repeating input;
                if $I$ only appears in loops
                then mark $I$ as a conditional and repeating input;
                if $I$ does not meet any condition above
                then mark $I$ as a singular input;
            end;
    end (the algorithm)

Another algorithm, Construct-Output-Port-For-Process-Network, is the same
as the one above except that it deals with send events to the external outputs rather than the receive events, and an output port structure is constructed for
the process network.

The algorithms build the canonical port structures directly. Since the port
structures for the higher level process is also transformed into the canonical
form, it is convenient to compare the canonical ports to check if the process
network receives and sends the same inputs and outputs as the decomposed
process.

In Figure 4.14 we construct the port structure through the STD which has
eight possible execution paths in Figure 4.10. In comparing the constructed
ports with that of the original process given in Figure 3.23 and the canonical
form in Figure 3.29, we find that the results are the same. It suggests that the
decomposition is correct.
The input side

Path 1:
P1.1: receive_i1 (f1)
P1.3: receive_i3 (f2)

Path 2:
P1.1: receive_i1 (f1)
P1.3: receive_i3 (f2)

Path 3:
P1.1: receive_i1 (f1)
P1.3: receive_i3 (f2)

Path 4:
P1.1: receive_i1 (f1)
P1.3: receive_i3 (f2)

Path 5:
P1.1: receive_i1 (f1)
P1.3: receive_i3 (f2) (within a loop)

Path 6:
P1.1: receive_i1 (f1)
P1.3: receive_i3 (f2)
P1.3: receive_i3 (f2) (within a loop)

Path 7:
P1.2: receive_i1 (f3)

Path 8:
P1.2: receive_i1 (f3)

The output side

Path 1:
P1.4: send_o1 (f4)

Path 2:
P1.4: send_o2 (f5)

Path 3:
P1.4: send_o1 (f4)

Path 4:
P1.4: send_o2 (f5)

Path 5:
P1.4: send_o1 (f4)

Path 6:
P1.4: send_o2 (f5)

Path 7:
P1.4: send_o1 (f4)

Path 8:
P1.4: send_o2 (f5)

Figure 4.14: Construction of the port structures through the STD in Figure 13
4.6 Brief summary

In this chapter we have presented a set of methods to check the consistency of system models. Since we work with an integrated modeling framework, the task of checking the consistency has been divided into several parts — the completeness, the consistency and the constructivity, for each of which a specific method is suggested.

One of our major considerations is to fill the gap between the theoretical approaches and the practical methodologies. This is particularly taken into account in the work on consistency checking. Hence, our approach in this chapter concentrate more on the feasibility of the methods than on their theoretical thoroughness. There are quite a few problems that we leave for further research. Some of them are:

- On consistency checking of database schemas, we limit our work to the scope of static constraints. Within the context of DB schema design and analysis there are certainly more constraints on the dynamic development of database states. For instance, Kung’s work aimed at providing a temporal framework for systems specification and verification by means of operation analysis and verification of temporal constraints [83]. However, from the practical point of view, we may meet two big problems: 1) how to treat all legal database states when we do not have a very small DB schema; 2) how to specify all the operations with specific parameters. Both problems are difficult to overcome at the design phase, so many approaches just treat them dynamically when an operation to be done on a database. As an example, see the approach with the NIAM method [111]. Therefore, in our modeling approach, we only provide static constraints and their associated verifications.

- On the checking of constructivity of the process hierarchies, the work has to be done with incomplete information, thus some assumptions instead of the detailed and precise descriptions about the behavior of the processes are used as the arguments to construct the STD. Therefore, the result we get from the algorithms might not be 100 % correct, although our method is still helpful in detecting many wrongly illustrated process diagrams. Since more detailed information is obtained when the analysis or design work reaches the bottom level where the process logic is to be specified explicitly, it seems that our method should be extended to a two-direction check method, both top-down and bottom-up directed, to exploit all information we can have.

Another problem is that the proposed method can only be applied to simple process networks instead of any possible topological structure. When the network is complex, the possible state vectors may soon become too large to be kept for the further calculation. This has been pointed out by Sindre. We have not started to develop a solution to that problem.
A possible solution, nevertheless, seems to be some "normal forms" of the network structure. We will do something on it. On the other hand, there are still some possible error states during the executions of a process network, e.g., the tight loop problem, ignored in our algorithms that focus on the major problems in consistency verification.
Chapter 5

PPP – a CASE tool to support the modeling

The PPP tool is the prototype of a CASE tool for building models of information systems. The modeling method, also called PPP, is an integrated modeling framework which includes Phenomenon Model (PM), Process Model(PrM), and Process Life Description (PLD) as its sub-models, and also supports the cross references among the components of the different sub-models. Since our modeling approach is a further development of PPP (e.g., ONE-R is an extension of PM and PPM is similar to PrM with some minor redefinitions), the work reported in the thesis is in fact a part of the PPP project. Therefore, a short introduction to the PPP tool is given in this chapter.

5.1 The working environment of PPP

PPP is developed on Sun stations with the software platform Sunview. Two supporting software systems are used to construct PPP: The BIM Prolog system and PCE — an object-oriented style graphical interface appended to Prolog systems.

PCE has been developed at the University of Amsterdam (SWI) since 1986 [7, 153]. It has a set of built-in classes used to create and manipulate graphical objects such as windows, pictures, bitmap-based graphics, buttons, pop-up menus, etc. These classes have provided facilities for the users to specify most model elements with some kind of graphical notation, thus results in a very user-friendly graphical interface of PPP.

PPP was written in BIM-prolog, since a logical programming language provides an excellent reasoning mechanism and convenient management of system specifications be stored as the facts.
5.2 The structure and repository of PPP

The overall structure of the PPP tool is illustrated in Figure 5.1 which shows the basic modules of PPP and its interactions with the users and the UNIX system.

Figure 5.1: The overall structure of the PPP tool

- The User Interface is set up by creating a set of PCE objects such as dialog boxes with labels and buttons, menu-items, and pictures on which the PPP elements can be drawn;

- The Drawing Editors provide the facilities for users to draw the elements in PM, PrM, PLD and their cross references.

- The Consistency Check can be executed automatically when a user is building a model by drawing the figures for the conceptual elements. For instance, when he connects a flow from one store to another store, then the flow is deleted automatically because a flow linking two stores is not allowed in the PrM syntax. The module can also be invoked explicitly by a user to check the whole specification of a system;

- The module Code Generation & Validation is able to transform a PLD diagram which describes the process logic of a bottom level process into
5.3. The modeling facilities in PPP

a program module written in ADA or C, and then execute the program to validate it.

- The Repository Management is, cooperating with the Configuration Management which traces the version developments of a system model and interchanges the system code or the model facts between the main memory and the UNIX file system.

5.3 The modeling facilities in PPP

The overall modeling framework of PPP is shown in Figure 5.2 that is quite similar to that given to SSADM (see Figure 2.5).

Figure 5.2: The overall structure of the PPP Modeling Framework

This framework is supported by a PPP window (shown in Figure 5.3) for its modeling framework, so that a user can:

- create a new system model or edit an existing model;
- create and edit a process diagram that describes the processes in the system. Each process can be further decomposed;
- create and edit a scenario that describes a piece of the reality whose information is to be managed within the system.

In Figure 5.4 we show the window of the PM editor. A user can edit a scenario by using a set of pre-defined graphical symbols on the left side of the window and use the mouse to locate the position of the element on the picture. Furtherly, a set of menu bar on the top of the window enables a user to

- store the scenario to a file, or load the scenario from a file;
Figure 5.3: The PPP window for its modeling framework

- check the correctness of the scenario. On the implemented version of PPP, the checking is still limited to the syntax level;

- link the class to the components in a PrM diagram (cross reference);

- generate the SQL code (create table statements) from the scenario to validate the scenario.

In Figure 5.5 we show the window of the PrM editor. Like that illustrated above for the PM editor, a user can draw the PrM (PPM) elements with the pre-defined graphical symbols and the mouse operations. A feature of the PrM editor is that the port structure can be formed by the pop-up menu defined on the processes directly and the menu also contains an item to enable a process to be decomposed into a new PrM diagram, like that shown in Figure 5.6.

The consistency check of a PrM diagram consists of two parts:

- syntax and static constraints check;

- dynamic check implemented with the STD method based on Sindre's work [8].

The code generation is invoked when a process has been at the bottom level
Figure 5.4: The PM window for drawing, checking and validation
Figure 5.5: The PrM window for drawing, checking and validating
Figure 5.6: A process is decomposed into a new PrM diagram
and has been given a Process Life Description (PLD) that has the following primitives:

- Start;
- Assignment;
- Choice;
- Iteration;
- Send;
- Receive.

All those component has a graphical notation. Therefore, when a process is illustrated by a PLD diagram, its input and output flows will be linked to the send and receive elements while the process logic is expressed by other elements. The code generation can work on the PLD and generate a program in ADA or C, so that the process logic can be validated. The detail of PLD is given in [61, 136].
Part III

The evaluation of the new approach
In the last part of the thesis, we try to validate the modeling approach presented in the previous part, and summarize the work.

In order to validate the modeling framework, we build a system model for the IFIP conference example which has been used to test many different methods. The system model has been given in quite detailed levels so that the potential application capabilities of our approach is shown.

Further, we give some comments relative to the evaluation criteria that we proposed in earlier part of the thesis, so the both the good results of our work and the still-left weak points are revealed. Finally, we mention some work to be done in the future.
Chapter 6

A case study

In order to validate the proposed modeling approach, we make a case study in this chapter. The example given here is the famous “IFIP working conference”, which has been used in chapter 3 but not as a complete example. Now we shall build a more complete system model to test if our approach works.

The example was proposed in as early as 1982 and has been used to illustrate many different modeling approaches. For the earliest version of PPM, an internal report containing a solution to the example was developed [126]. Because the work reported in this thesis is an evolution of the early version of PPM, we also provide a solution to the same example. This will make it possible to compare the new approach not only with other modeling approaches, but also with its predecessors as presented in [125, 10, 108, 110, 61] respectively.

The system model built in this chapter has many similarities with that given in [126] because both the data model and the process model in COMIS are developed directly from old PM and PPM. However, it is worth mentioning some differences between the two case studies so that some reasons for making the efforts to develop the modeling framework in the thesis are revealed:

- In [126] the concept entity class is used both in the data model as objects whose data are to be managed and in the process model where it is regarded as the external entities that interact with processes. No distinction between the roles of active objects and passive objects is made. Based on the principle given in the previous chapters, however, our system model makes the distinction explicitly, even though some entities have dual roles in the system. For instance, the entities “referees” are regarded as external agents in the process description, but are also treated as data entities in the data type description. By the way, we know clearly that the first role means that the referees are active in the system with some definable behaviors, whereas the second one means that the data about them are managed in the system as well;
In [126] the external entities interchange information with the process by means of "document flow", i.e., the physical information carrier, and signal. In our case study, we only consider the information as carried by data in forms of data records, text, signal, etc.; in other words, only data flows linking system components are considered and any kind of material flow among them is ignored;

[126] shows a way of specifying the dynamic aspect of a system: first define the high level processes; then decompose the processes step by step until the bottom level; finally the process logic for any ground level process can be specified. This is a typical way for specifying a system in the DFD approach [39, 59, 157] and is also adopted by the Activity and Behavior Modeling approach [86]. However, as we have pointed out, in an information system, there usually exist some processes whose logics cannot be specified (e.g., a decision making process) or need not to be specified (e.g., the execution of a existing software), so not every ground level process needs a process logic description. In order to show the point, we provide a solution where only these two kinds of ground level processes appear in the system. We have also introduced processing resources in our solution, and we will see that the work for the activities of the IFIP working conference may be done by the members of PC and OC, and by some software packages that are available in market.

6.1 Problem description

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

The management of such a conference is usually done by two cooperating committees. The program committee (PC) handles the contents of the conference, say, the reviewing of papers, comprising sessions and tutorials, etc. On the other hand, the organizing committee (OC) handles the administration work, e.g., sending out invitation, registration of attendants, arrange time and places for sessions, deal with financial problems, etc.

The management of a conference comprises many activities with a lot of data, signals, documents and material being interchanged among people and supporting systems. However, since the task of information systems is to support the information processing of these activities, the information system model will usually consists of components that reflect the information processing activities.

According to our modeling framework shown in Figure 3.3, we should build a
system model at two levels – the business system level and the technical system level. At the business level, we specify the major functions of the system and the types of data to be managed in the activities that support the system's (business) functions; at the technical system level, we specify the application systems consisting of executable programs called actors with various technical functions. Meanwhile, through the cross references, we specify how the data are manipulated in the system activities (processes) and how the actors execute in order to support some parts of the activities.

Our system model consists of the following parts:

- The system's (business) functions: naturally we define two system functions by the work contents of the two committees: **programming** and **organizing**:

  1. **programming** is realized with following specific activities (processes):
     - planning & announcing the conference;
     - recording responses;
     - distributing papers to referees;
     - receiving referee's reports;
     - determine the acceptances of the papers;
     - grouping the papers into sessions.

  2. **organizing** is realized with following specific activities (processes):
     - preparing the conference;
     - sending out invitations;
     - recording confirmations of attendants;
     - receiving payments for the conference;
     - generating the arrangement for the conference;
     - registration;
     - dealing with reserving hotel rooms and booking tickets;

- The scenarios: we define two basic scenarios:

  1. **scenario 1** contains the entity classes and relationship classes mostly occurring in the activities of programming function, such as persons, authors, referees, papers and their relationships;

  2. **scenario 2** contains the entity classes and relationship classes mostly occurring in the activities of organizing function, such as attendants, confirmations, payment notes, orders for hotel room or tickets, and so on.

- The Application systems and the comprising actors: we define two application systems **programming supporting system** and **organizing supporting system**; both consist of the following actors:

  1. **Ar1**: a data management system with screen form interface;
2. **Ar2**: a report generator which can fetch a set of data from a database and generate a formatted report.

## 6.2 The data model

### 6.2.1 The basic scenarios

With the concepts provided by ONE-R, we describe the data objects to be managed in two basic scenarios; each basic scenario contains some entity classes and relationship classes that are of particular interest to the system functions programming and organizing.

Shown in Figure 6.1, *Scenario* contains the entity classes and relationship classes that are mainly occurring in the activities of the program committee:

- **persons**, including the sub-classes **referees**, **journal editors** and **potential participants**, with following attributes:
  
  - **name**: the attribute has a value of the composite data type **name** that has three components **first name**, **middle name**(not mandatory) and **last name**;
  
  - **address_information**: the attribute also has a value of a composite type named **address_information** that has components **address**, **country** and non-mandatory components **phone**(may be more than one) and **email**;
  
  - **research_directions**: the attribute defined specially for **referees** is a set of words of research or work fields that shows a referee's ability to review some kinds of papers;

- **journals**. Information about journals is needed, especially when no information about their editors can be obtained. For instance, some network news group can be regarded as a journal in which the conference can also be advertised. The attributes for the class are:
  
  - **name**: the name of a journal, which is simply of the type **string**;
  
  - **address_information**: which has the same data type with that for **person**;

- **papers** are reviewed by **referees** and grouped into **sessions**. The relevant attributes are:
  
  - **number**: an **integer** number is assigned to each paper;
  
  - **title**, **written_language**, **keywords**, **author's_names** are all defined for **paper**;


Figure 6.1: Scenari0: the entity classes and relationship classes processed by the program committee
for the relationship class review, several attributes are introduced in order to reflect the result of the referee’s evaluation to the paper: comment is a piece of text, evaluation is a composite datum consisting of the components originality, technique_quality, relevance_to_the_conf. and representation, and overall_rating shows the referee’s opinion about the acceptance of the paper;

- the class session has the attributes number, name and chairman. The last attribute is of the type person;

In Figure 6.2, Scenario2 contains the entity classes and relationship classes mainly occurring in the activities of the organizing committee:

- persons, including the sub-classes potential participants and confirmed participants with following attributes:
  - name;
  - address_information;

The data types of them have been defined earlier;

- payments, confirmed participants and the relationship class pay with following attributes:
  - number: the attribute has an integer value assigned to each payment;
  - amount: the attribute records the amount of each payment;
  - paying_for: the attribute describes the manner that a participant make his payment, e.g., cash, money order, check, etc.;

- accepted papers grouped into sessions. Their attributes are:
  - number, title, written_language, author’s_names for the class accepted paper as that described above;
  - number, name and chairman for session;

- hotels and transportation services and orders from the participants for the services. The classes have following attributes:
  - name and address_information are defined for hotels and transportation services;
  - item is also an attribute defined for the two classes, that is of a composite type with two components service and price;
  - the class order for service has two attributes number and contents that is a set of sentences for different items of services.
Figure 6.2: Scenario2: the entity classes and relationship classes processed by the organizing committee
In the two scenarios, some constraints on the data have been expressed with the structural components. For instance, the participation types of the entity classes in the relationship types express some constraints. In \textit{Scenario}$_1$, the type \textit{full}$_n$ appears on the two sides of the relationship class \textit{write}, so neither an author nor a paper can be recorded in the database independently of the existence(s) of its corresponding object(s) in the other class. However, although a paper must be reviewed by some referees, since the reviewing relationship between a paper and its referees can only be determined some days after it was submitted, the data about a paper should be recorded without considering of its relationship with the referees, thus the participation type of the class \textit{paper} in the relationship class \textit{review} is \textit{partial}$_n$. More constraints are defined in similar ways, based on various considerations.

### 6.2.2 The external scenarios

The external scenarios are built for the specific processes in which some parts of the basic scenarios are to be accessed. We should claim here that the external scenarios can only be created after the process model given in the next section is built, because the contents of them can only be known when the processes have been well defined. Why we illustrate the external scenarios in this section is just because we need to introduce the applications of sub-models of our modeling framework separately.

Now we build these external scenarios by the following SDL statements:

```plaintext
build_external_scenario Initial_data
   referencing Scenario_1
   accessible classes:
       person, referee, journal, paper,
       editor, work_for;
end

build_external_scenario Intention
   referencing Scenario_1
   accessible classes:
       potential_participant, author, paper,
       write;
end

build_external_scenario Paper_referees
   referencing Scenario_1
   accessible classes:
```
6.2. The data model

    paper, referee,
    review
end

build_external_scenario Selection_of_papers
  referencing Scenario_1
  accessible classes:
    paper, accepted_paper, rejected_paper;
end

build_external_scenario Session_papers
  referencing Scenario_1
  accessible classes:
    accepted_paper, session, grouped_in;
end

build_external_scenario Potential_participant
  referencing Scenario_2
  accessible classes:
    potential_participant
end

build_external_scenario Services
  referencing Scenario_2
  accessible classes:
    hotel, transportation_service;
end

build_external_scenario Session
  referencing Scenario_2
  accessible classes:
    accepted_paper, session, grouped_in;
end

build_external_scenario Confirmation_payment_registration
  referencing Scenario_2
  accessible classes:
potential_participant, confirmed_participant, 
registered_participant, payment, pay;
end

build_external_scenario Order_for_service 
referencing Scenario_2
accessible classes:
  confirmed_participant, order_for_service, order_for_hotel,
  reserve_room_of, order_for_ticket, book_ticket_of;
end

6.3 The process model

6.3.1 The top level processes

The two system functions are realized with a set of processes, i.e., the specific activities are invoked at different time points under particular conditions. By the PPM modeling concepts, we get two top level process network diagrams (PNDs), each of which contains the processes to support the system functions programming and organizing respectively.

Figure 6.3 is the PND programming.d0 with more information given in Figure 6.4. The PND contains the following processes:

1. process P1: plan and announce the conference
   • inputs:
     - $f_1$: a signal to start the process.
   • outputs:
     - $f_2$: a person's information to be kept for further use;
     - $f_3$: a referee's information;
     - $f_4$: information about a journal and/or its editors (with the relationships between them);
     - $f_5$: the announcement of the conference, including the call for papers;
     - $f_6$: the news about the conference to be sent to journals;
     - $f_7$: a signal showing that the process has terminated.
   • function:
     The process is triggered by a signal issued from the chairman of the program committee, then two tasks are finished with the execution
Figure 6.3: PND *programming.d₀*: the processes executed in the program committee
AGENTS AND THEIR TRIGGERING RULES

A1: chairman of the program committee
TR1: WHEN he wants to start P1 THEN send f1;
TR2: WHEN the deadline time for submitting papers is due
     THEN send f13;
TR3: WHEN the deadline time for receiving referee's reports
     is due THEN send f23;
TR4: WHEN P5 has been executed THEN send f29;

A2: person to be informed
A3: journal to be informed
A4: potential participant
TR1: WHEN he wants to attend the conf. and submit a paper
     THEN send f13, f15;
TR2: WHEN he wants to attend the conf. THEN send f13;
A5: referee
TR1: WHEN he finishes the reviewing work THEN send f18, f19;

DATA IN FLOWS
f1: a signal showing the work of panning and announcing
    the conference is to be started
f2: person's infor
f3: referee's infor
f4: infor. about journal, journal editor and the relationship between
    them
f5: announcement of the conference
f6: the news about the conference
f7: a signal showing that the work of panning and announcing
    the conference is finished
f8: a potential participant's message for will of attending
f9: infor. about a submitted paper
f10: infor. about potential participant
f11: infor. of author, paper and the relationship between them
f12: a signal showing that the recording work is done
f13: a signal to begin the work to distribute submitted papers to
    the referees
f14: infor. about a referee
f15: infor. about papers, authors and the relationship between them
f16: infor about papers, referees and the relationship between them
f17: a signal showing that the distributing work is finished
f18: a referee's message for having reviewed some papers
f19: the referee's report for reviewing a paper
f20: infor. about relationship between paper and referee
f21: infor about the relationship between a paper and a referee with the
    reviewing result
f22: a signal showing that the recording work is done
f23: a signal to start selecting the papers
f24: infor. about a paper and its relationship with its referees
f25: message of accepting a paper
f26: message of rejecting a paper
f27: infor. about a paper's acceptance or rejection
f28: a signal showing that the selecting work is finished
f29: a signal to start the work of grouping sessions
f30: infor. about an accepted paper
f31: infor. about a session and its relationship with accepted papers
f32: a signal showing that the grouping work is finished

I/O CONDITIONS

<table>
<thead>
<tr>
<th>P1</th>
<th></th>
<th>P2</th>
<th></th>
<th>P3</th>
<th></th>
<th>P4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f1</td>
<td></td>
<td>f8</td>
<td>f9</td>
<td>f13</td>
<td>f14</td>
<td>f15</td>
</tr>
<tr>
<td>f7</td>
<td>X</td>
<td>f10</td>
<td>X</td>
<td>f16</td>
<td>X</td>
<td>X</td>
<td>f18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f11</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>f19</td>
</tr>
<tr>
<td>f8</td>
<td></td>
<td></td>
<td></td>
<td>f21</td>
<td>f18</td>
<td>f19</td>
<td>X</td>
</tr>
<tr>
<td>f9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 6.4: More information about the components in programming.d0
of the process: collecting necessary information into the data store for further processing and sending the announcements and news about the conference to the people and journals to attract potential authors and potential participants. When the process terminates, a signal is also sent to the chairman of the program committee to inform him of that the tasks of the process have been finished.

2. process P2: record response

- **inputs:**
  - \( f_8 \): a message from a person showing that he is interested in attending the conference;
  - \( f_9 \): the information about a submitted paper.

- **outputs:**
  - \( f_{10} \): information about a potential participant;
  - \( f_{11} \): information about the submitted paper, the author(s) of it, and the relationships between them;
  - \( f_{12} \): a signal showing that the information about a response has been recorded.

- **function:**
  The process is triggered by a response message from a potential participant who also possibly submits a paper to the conference. During the process, if the person and his paper has not been recorded yet, then the information about the person and his paper is sent to a data store. A signal is sent out when the process terminates but nobody is informed about it for that the process is a routine work.

3. process P3: distribute papers

- **inputs:**
  - \( f_{13} \): a signal to start the process;
  - \( f_{14} \): information about a referee;
  - \( f_{15} \): information about a paper and its author(s).

- **outputs:**
  - \( f_{16} \): information about the relationships between papers and referees;
  - \( f_{17} \): a signal showing that the process has terminated.

- **function:**
  The process is triggered by a signal from the chairman of the program committee; it determines which papers will be reviewed by which referees. The process uses the information about the referees and the papers, and send the information about the reviewing relationships between referees and papers to a data store. When the process terminates, a signal is sent to the chairman to inform him that the distribution task has been finished.

4. process P4: record referee's report
• inputs:
  – $f_{18}$: a message from a referee to deliver his reviewing results;
  – $f_{19}$: the report about a paper;
  – $f_{20}$: information about the relationship between a paper and a referee.

• outputs:
  – $f_{21}$: information about the evaluation of a paper given by a referee;
  – $f_{22}$: a signal showing that the report information has been recorded.

• function:
The process is triggered by a referee's message with a set of reports for the papers that the referee reviewed. The process receives the report and sends the information to a data store, and modifies the data about the relationships between referees and papers. When the process terminates, a signal is given but nobody is informed about it because this is not needed!

5. process P5: determine the acceptance of papers

• inputs:
  – $f_{23}$: a signal to start the process;
  – $f_{24}$: information about a paper and the evaluation to it given by referees.

• outputs:
  – $f_{25}$: a acceptance message;
  – $f_{26}$: a rejection message;
  – $f_{27}$: information about a paper's acceptance or rejection;
  – $f_{28}$: a signal showing that the work of selecting papers has terminated.

• function:
The process is triggered by a signal from the chairman of the program committee. It determines the acceptance or rejection of every submitted paper, according to the evaluations given to the papers. Each author is given a message about his paper and the information is also sent to a data store. When the process terminates, the chairman is sent a signal to inform him of that the selection work is finished.

6. process P6: group accepted papers into sessions

• inputs:
  – $f_{29}$: a signal to start the process;
  – $f_{30}$: information about an accepted paper.

• outputs:
6.3. The process model

- \( f_{31} \): information about a session and its relationships with the accepted papers;
- \( f_{32} \): a signal showing that the grouping work has terminated.

**function:**
The process is triggered by a signal from the chairman of the program committee. It decides the names and contents of the session held during the conference and group the accepted papers into sessions. The information about the sessions is sent to a data store and finally the chairman of the committee is informed when the work of the process finishes.

In the same way, Figure 6.5 is the PND organizing \( d_0 \) with more information given in Figure 6.6. The PND contains following processes:

1. process \( P_1 \): prepare the conference
   - **inputs:**
     - \( f_1 \): a signal to start the process;
     - \( f_2 \): information about a potential participant;
     - \( f_3 \): information about a session and its papers;
     - \( f_4 \): information about a hotel or a transportation service;
   - **outputs:**
     - \( f_5 \): information about a potential participant;
     - \( f_6 \): information about a hotel or a transportation;
     - \( f_7 \): information about a session and its papers;
     - \( f_8 \): a signal showing that the process has terminated.
   - **function:**
The process is triggered by a signal from the chairman of the organizing committee. It only prepares some necessary information for organizing the conference. The information does not need to be collected because the program committee, hotels and transportation services have already provided all the information; the secretary need only input these data into corresponding data stores for later use. When the process terminates, the chairman will receive a signal to inform of that the preparation work is finished.

2. process \( P_2 \): send out invitations
   - **inputs:**
     - \( f_9 \): a signal to start the process;
     - \( f_{10} \): information about a potential participant;
     - \( f_{11} \): information about a hotel or transportation service;
   - **outputs:**
     - \( f_{12} \): an invitation message;
     - \( f_{13} \): a signal showing that all invitations have been sent out.
Figure 6.5: PND organizing.d0: the processes executed in the organizing committee
AGENTS AND THEIR TRIGGERING RULES

A1: chairman of the organizing committee
TR1: WHEN he wants to start P1 THEN send f1;
TR2: WHEN P1 is finished and he wants to start P2 THEN send f9;
TR3: WHEN the date to arrange the conference is due THEN send f21
is done IF P4 is idle THEN send f23;
A2: secretary
A3: potential participant
A4: confirmed participant
TR1: WHEN he decides to attend the conf. THEN send f14
TR2: WHEN he is ready to pay THEN send f17;
TR3: WHEN he wants to reserve rooms or book tickets THEN send f28 or f29;
A5: arrived participant
TR1: WHEN he arrives THEN send f24;

DATA IN FLOWS

f1: a signal showing the work of preparing the conference is to be started
f2: infor. about a potential participant
f3: infor. about a session and its papers
f4: infor. about a hotel or transportation service
f5: infor. about a potential participant
f6: infor. about a hotel or transportation service
f7: infor. about a session and its papers
f8: a signal showing that the preparing work is finished
f9: a signal showing that the invitations are to be sent out
f10: infor. about potential participant
f11: infor about a hotel or transportation service
f12: an invitation
f13: a signal showing that all the invitations have been sent out
f14: a message confirming attendance
f15: infor. about a confirmed participant
f16: a signal showing that a confirmation message is recorded
f17: a message showing that a payment has been made
f18: infor. about a confirmed participant
f19: infor. about a payment and its relationship with a confirmed participant
f20: a signal showing that a payment is recorded
f21: a signal showing that the work to make the arrangement of the conf. is to be started
f22: infor. about a session and its papers
f23: an arrangement for the conf.
f24: a message of an arrived participant
f25: infor. about a confirmed participant
f26: infor. about a registered participant
f27: a signal showing a participant is registered
f28: a filled order
f29: a message for ordering
f30: a item
f31: infor. about a hotel or transportation service
f32: infor. about a reservation
f33: infor. about a booking action
f34: a reply to the request

I/O CONDITIONS

<table>
<thead>
<tr>
<th>P1:</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
</tr>
</thead>
<tbody>
<tr>
<td>f5</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f6</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>f7</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P2:</th>
<th>f10</th>
<th>f11</th>
<th>f12</th>
<th>f13</th>
<th>f14</th>
</tr>
</thead>
<tbody>
<tr>
<td>f14</td>
<td></td>
<td></td>
<td></td>
<td>f14</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P3:</th>
<th>f15</th>
<th>f16</th>
<th>f17</th>
<th>f18</th>
</tr>
</thead>
<tbody>
<tr>
<td>f15</td>
<td></td>
<td></td>
<td>f17</td>
<td>f18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P4:</th>
<th>f19</th>
<th>f20</th>
<th>f21</th>
<th>f22</th>
</tr>
</thead>
<tbody>
<tr>
<td>f19</td>
<td></td>
<td></td>
<td>f21</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P5:</th>
<th>f22</th>
<th>f23</th>
</tr>
</thead>
<tbody>
<tr>
<td>f22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f23</td>
<td></td>
<td>f25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P6:</th>
<th>f26</th>
</tr>
</thead>
<tbody>
<tr>
<td>f26</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P7:</th>
<th>f28</th>
<th>f29</th>
<th>f30</th>
<th>f31</th>
</tr>
</thead>
<tbody>
<tr>
<td>f28</td>
<td></td>
<td></td>
<td>f30</td>
<td></td>
</tr>
</tbody>
</table>

| P8: | f32 | f33 | f34 |
|-----|-----|-----|
| f32 |     |     |
| f33 |     |     |
| f34 |     |     |

Figure 6.6: More information about the components in organizing.d0
• function:
The process is triggered by a signal from the chairman of the organizing committee. During the process, an invitation to the conference is sent to each potential participant. When the process terminates the chairman receives a signal informing him of that all invitations have been sent out.

3. process P3: record confirmation for attending

• inputs: a confirmation message from a potential attendant.
  - $f_{14}$;

• outputs:
  - $f_{15}$: information about a confirmed participant;
  - $f_{16}$: a signal showing that a confirmation is recorded.

• function:
The process is triggered by a message from a potential participant to confirm that he wants to attend the conference. The process records the information in a data store. When the process terminates, a signal is given but nobody needs it because there is no subsequent work to be done.

4. process P4: receive payment

• inputs:
  - $f_{17}$: a message showing that a payment has been made;
  - $f_{18}$: information about a confirmed participant

• outputs:
  - $f_{19}$: information about a payment and its relationship to a confirmed participant;
  - $f_{20}$: a signal showing that a payment is recorded.

• function:
The process is triggered by a message sent from a confirmed participant showing that he has paid for the conference. The information about the payment is recorded into a data store and a signal showing that the recording work has been done is sent out when the process terminates.

5. process P5: generate arrangement of the conference

• inputs:
  - $f_{21}$: a signal to start the work;
  - $f_{22}$: information of a session and its papers.

• outputs:
  - $f_{23}$: an arrangement for the time and places of the sessions.
6.3. The process model

• function:
  This process is triggered by a signal from the chairman of the organizing committee. The task of the process is to produce an arrangement, i.e., an agenda for the conference, particularly the times and places of the sessions. The information about the sessions and its papers is used during the process. When the process terminates, the generated arrangement is sent to the secretary.

6. process P6: register attendant

• inputs:
  – $f_{24}$: a message from a arrived participant;
  – $f_{25}$: information about a confirmed participant.

• outputs:
  – $f_{26}$: information about the registered participant;
  – $f_{27}$: a signal showing that the registration work has been done.

• function:
  The process is triggered by a message about an arrived participant. The process just lets the participant register and puts the registration information into a data store. A termination signal is sent out but not used because there is no subsequent work to be done.

7. process P7: reserve room or book ticket

• inputs:
  – $f_{28}$: information about a filled order;
  – $f_{29}$: a message for ordering;
  – $f_{30}$: information about an item;
  – $f_{31}$: information about a hotel or a transportation service.

• outputs:
  – $f_{32}$: information about a reservation;
  – $f_{33}$: information about a booking;
  – $f_{34}$: a reply message to the request.

• function:
  The process is triggered by a participant’s (confirmed or registered)’s filled order or a message (telephone call, dialog or other forms) for ordering. The process handles the request for reserving rooms in hotels or booking tickets in the transportation services. Whatever the result is, say, the order is accepted or rejected, the process terminates by sending the participant a message about the result; however, if the process succeeds in reserving or booking, the information about the reserving and booking is also sent to a data store during the process.
6.3.2 The decomposition of the processes

The processes are all on the top level and therefore can be further decomposed into sub-processes until a bottom level where no further decomposition is possible or necessary. We only show the decompositions of processes P1, P2 and P3 of the system function programming that is carried by the program committee. However, the decompositions have been detailed enough to reach the bottom level for all of them, so that how the system activities are supported by the technical systems (actors) can be described clearly in later sections.

The decomposition of process P1

In figure 6.7 the decomposition of process P1 is shown. More information about the decomposition is given in Figure 6.8. In Figure 6.7.a, process P1 is decomposed to the PND programming.d1 which contains the following sub-processes:

- **P1.1**: prepare information
  The process collects and stores the initial data for further processing. It is triggered by a signal issued by the chairman of the programming committee (f1), and sends the information about some persons (to be informed about the conference), referees and journals and/or their editors to which the news about the conference is to be published into data store S1;

- **P1.2**: announce the conference
  After the termination of P1.1, process P1.2 is triggered. P1.2 use the address information from data store S1, sends a call for participation and contribution of papers to potential participants (at the moment all persons whose information stored in S1 are regarded as potential participants), as well as the news about the conference to some journals. When P1.2 terminates, it sends a signal to the chairman to tell him about the end of P1.

Since the two processes are still not at the bottom level, they are further decomposed. In Figure 6.7.b, process P1.1 is decomposed to the PND programming.d1,1 which contains following the sub-processes:

- **P1.1.1**: collect information
  The process is the first part of P1.1. It collects all the necessary initial data (we may guess that some people search the data from various sources) and generates a list of such data for further processing;

- **P1.1.2**: edit data items
  The process edits the collected data, transform them into formatted data
6.3. The process model

Figure 6.7: The decomposition for P1
Chapter 6. A case study

DATA IN FLOWS
f1.1: a signal showing the work of prepare information has ended
f1.2: infor. about the address or email of an person or journal

I/O CONDITIONS

<table>
<thead>
<tr>
<th>f1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3</td>
</tr>
<tr>
<td>f6</td>
</tr>
</tbody>
</table>

a) more information about the components in PND programming.d_1

DATA IN FLOWS
f1.1.1: a list of collected names, addresses, telephone numbers, email addresses, etc.
f1.1.2: infor. about a person
f1.1.3: an insert command
f1.1.4: infor. about a referee
f1.1.5: an insert command
f1.1.6: infor. about journal & editors
f1.1.7: an insert command
f1.1.8: a signal showing that the editing has ended
f1.1.9: a signal showing that keeping infor. for persons has ended
f1.1.10: a signal showing that keeping infor. for referees has ended
f1.1.11: a signal showing that keeping infor. for journals & editors has ended

I/O CONDITIONS

<table>
<thead>
<tr>
<th>f1.1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f1.1.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>f4</td>
</tr>
</tbody>
</table>

b) more information about the components in PND programming.d_1.1

DATA IN FLOWS
f1.2.1: a query command
f1.2.2: a signal to start sending calls & news
f1.2.3: a list of address

I/O CONDITIONS

<table>
<thead>
<tr>
<th>f1.2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1.1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f1.2.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>f5</td>
</tr>
<tr>
<td>f6</td>
</tr>
</tbody>
</table>

c) more information about the components in PND programming.d_1.2

Figure 6.8: More information for the decomposition of P1
items, and triggers the processes for putting data of different classes (person, referee and journal and editor) into data stores. When the process ends, it sends out a signal showing that the edit work is finished;

- **P1.1.3**: keep information about persons
  This process does the work that can be supported by a technical system. It is triggered by a request from process **P1.1.2**, receives edited data about a person repeatedly, and sends the data into data store **S1**. When it ends it gives a signal;

- **P1.1.4**: keep information about referees
  The process does the same work as **P1.1.3** but the data it receives and sends are about the referees;

- **P1.1.5**: keep information about journals and editors
  The process does the same work as **P1.1.3** but the data it receives and sends are about the journals and/or their editors;

- **P1.1.6**: inform the end of editing
  This process is actually “artificial” since no work is needed for it. It appears here as a process just for indicating that when process **P1.1.3**, **P1.1.4** and **P1.1.5** have all terminated, the signal in **P1.1** can be sent out to inform about that the work of **P1.1** has been done.

In Figure 6.7.c, process **P1.2** is decomposed to the PNDprogramming.d1.2 which contains following sub-processes:

- **P1.2.1**: start to fetch address information
  This process is triggered by the signal showing that the work of process **P1.1** has ended so that the work of announcing the conference can start. It sends a request to process **P1.2.2** to start fetching address information;

- **P1.2.2**: get address information
  The process does the work that can be supported by a technical system. It is triggered by a request from process **P1.2.1** and fetches the address information repeatedly from the data store **S1** and then sends the data out for further processing;

- **P1.2.3**: send out call and news
  This process is triggered by the signal from **P1.2.2** saying that all address information has been sent out. It then repeatedly sends out “call for paper” and news about the conference to the potential participant and journals respectively, according to the address data it receives. When the work has finished, it terminates by sending to the chairman a signal telling him that the work of process **P1** has finished.
Figure 6.9: The decomposition for P2
6.3. The process model

DATA IN FLOWS

f2.1: infor. about an potential participant
f2.2: a signal showing no need for recording
f2.3: an insert command
f2.4: infor. about a potential participant
f2.5: an insert command
f2.6: infor. oabout a paper and its authors
f2.7: a signal showing that keeping infor. about a potential participant has ended
f2.8: a signal showing that keeping infor. about a paper and the authors has ended

I/O CONDITIONS

<table>
<thead>
<tr>
<th>P2.1:</th>
<th>P2.2:</th>
<th>P2.3:</th>
<th>P2.4:</th>
</tr>
</thead>
<tbody>
<tr>
<td>f9</td>
<td>f2.4</td>
<td>f2.6</td>
<td>f2.2</td>
</tr>
<tr>
<td>f2.6</td>
<td>f10</td>
<td>f11</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a) more information about the components in PND programming.d 2

DATA IN FLOWS

f2.1.1: a query command
f2.1.2: information about a potential participant
f2.1.3: a signal showing that the query has finished
f2.1.4: information about a potential participant or empty

I/O CONDITIONS

<table>
<thead>
<tr>
<th>P2.1.2:</th>
<th>P2.1.3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2.1</td>
<td>f2.1.2</td>
</tr>
<tr>
<td>f2.1.3</td>
<td>f9</td>
</tr>
<tr>
<td>f2.4</td>
<td>f2.1.4</td>
</tr>
<tr>
<td>f2.6</td>
<td>X</td>
</tr>
<tr>
<td>f2.2</td>
<td>X</td>
</tr>
</tbody>
</table>

b) more information about the components in PND programming.d 2.1

Figure 6.10: More information about the decomposition of P2
The decomposition of process P2

In figure 6.9 the decomposition of process P2 is given. More information for the decomposition are given in Figure 6.10. In Figure 6.9.a, process P2 is decomposed to the PND \textit{programming.d}_1 which contains the following sub-processes:

- \textbf{P2.1}: check information
  The process is triggered by a message from a potential participant saying that he wants to attend the conference. The potential participant may also have submitted a paper. The process checks if the intention has been recorded before; it is necessary, it edits the information about the potential participant and possibly information about papers in order to put the information into a data store. The output of the process may be simply a signal to show that the intention has been recorded before, or the edited data that are sent into the data store S2;

- \textbf{P2.2}: keep information about potential participants
  This process does the work that can be supported by a technical system. It is triggered by a request from process P2.1, receives edited data about a potential participant repeatedly, and send the data into data store S2. When it ends it gives a signal;

- \textbf{P2.3}: keep information about papers, authors and their relationships
  The process does the same work as P1.1.3 buy the data it receives and sends are about the authors, papers and their relationships;

- \textbf{P2.4}: inform the end of recording
  Like P1.1.6, this process is "artifical" since no work is needed. It appears here as a process just for indicating that when process P2.2 and P2.3 have all terminated, the signal in f12 can be sent out to inform about that the work of P2 has been done.

Since process P2.1 is not at the bottom level, in Figure 6.9.b it is further decomposed into the PND \textit{programming.d}_{2,1} which contains the following sub-processes:

- \textbf{P2.1.1}: start to check
  This process is triggered by a potential participant’s message. It invokes a query to check if the information about potential participant has been recorded before;

- \textbf{P2.1.2}: query information about the potential participant
  The process, possibly supported by a technical system, checks if a potential participant’s intention to attend the conference has been recorded in data store S2;
6.3. The process model

- **P2.1.3**: decide if to input
  This process decides if the information about the potential participant and/or his paper is to be put into data store S2. If it decides not to input, then it sends a signal in f2.2, otherwise it edits and sends out the data items about the potential participant and/or his submitted paper respectively.

The decomposition of process P3

In figure 6.11 the decomposition of process P3 is given. More information for the decomposition is given in Figure 6.12. In Figure 6.11.a, process P3 is decomposed to the PND *programming.d₃* which contains the following sub-processes:

- **P3.1**: list papers and referees
  The process is triggered by a signal from the chairman of the program committee. It fetches data about papers and referees and generates formatted reports that are sent to process P3.2 for further work;

- **P3.2**: assign tasks to referees
  The process decides which papers are to be reviewed by which referees. It receives the lists (reports) about the committed papers and sends out a list which assigns papers to referees;

- **P3.3**: record the information about the distribution
  The process puts the information about the "review" relationships between papers and referees into data store S3. When terminates, it also sends the chairman a signal to inform about that the work of P3 has been done.

Process P3.1 is not at the bottom level, so in Figure 6.11.b it is further decomposed into the PND *programming.d₃.₁* which contains following sub-processes:

- **P3.1.1**: start listing
  This process is triggered by the signal from the chairman to start the work of process S3. It then invokes process 3.1.2 and P3.1.3 by requesting them to produce formatted reports;

- **P3.1.2**: list referees
  This process does the work that can be supported by a technical system. It is triggered by a request from process P3.1.1, fetches data about referees from S1, and generates a formatted report. When it ends it gives a signal;

- **P3.1.3**: list papers
  Like P3.1.3, this process does the work that can be supported by a
Figure 6.11: The decomposition for P3
6.3. The process model

DATA IN FLOWS

f3.1: a list of referees
f3.2: a list of submitted papers
f3.3: a signal showing the work
of listing has ended
f3.4: a list assigning each paper to
a referee

I/O CONDITIONS

<table>
<thead>
<tr>
<th>P3.1:</th>
<th>f3.1</th>
<th>f3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3.1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>f3.2</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P3.2:</th>
<th>f3.1</th>
<th>f3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3.4</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>f3.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P3.3:</th>
<th>f3.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3.4</td>
<td></td>
</tr>
</tbody>
</table>

a) more information about the components in PND programming.d3

DATA IN FLOWS

f3.1.1: a signal to start listing referees
f3.1.2: a signal to start listing papers
f3.1.3: the format for listing referees
f3.1.4: the format for listing papers
f3.1.5: a signal showing that listing referees has finished
f3.1.6: a signal showing that listing papers has finished

I/O CONDITIONS

<table>
<thead>
<tr>
<th>P3.1.2:</th>
<th>f3.1.3</th>
<th>f3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3.1.3</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>f3.1</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P3.1.3:</th>
<th>f3.1.4</th>
<th>f3.1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3.1.4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>f3.1.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P3.1.4:</th>
<th>f3.1.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3.1.6</td>
<td></td>
</tr>
</tbody>
</table>

b) more information about the components in PND programming.d3.1

DATA IN FLOWS

f3.3.1: a command to insert infor.
f3.3.2: infor. about the "review" relationship between a paper and a referee
f3.1.1: a signal showing the end of editing

I/O CONDITIONS

<table>
<thead>
<tr>
<th>P3.3.1:</th>
<th>f3.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3.3.1</td>
<td>X</td>
</tr>
<tr>
<td>f3.3.2</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P3.3.2:</th>
<th>f3.3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3.3.2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f16</th>
<th>X</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>f3.4</th>
</tr>
</thead>
</table>

c) more information about the components in PND programming.d3.3

Figure 6.12: More information for the decomposition of P3
technical system. It is triggered by a request from process P3.1.1, fetches data about submitted papers from S1, and generates a formatted report. When it ends it gives a signal;

- **P3.1.4**: inform the end of listing
  This process is also an "artificial" process that no work is actually done. It only shows that once both process P3.1.2 and P3.1.3 terminate, a signal is generated showing the listing work has finished;

Also, in Figure 6.11.b process P3.3 is further decomposed into the PND programming.d3,3 which contains following sub-processes:

- **P3.3.1**: edit data items
  This process is triggered by the list of assignment between referees and papers. It edits the list and send out data items to send them to data store S3;

- **P3.3.2**: keep information about the “review” relationships
  This process does the work that may be supported by a technical system. It is triggered by a request, then receives data items and sends them into S3 repeatedly. When it terminates, it sends out a signal to the chairman of program committee, that shows the work of process P3 has finished.

### 6.4 The actor model

#### 6.4.1 The actors and their functions

There are two application systems – the **programming supporting system** and the **organizing supporting system** to provide technical means for supporting the two major functions of the IFIP conference. Both systems only consist of two actors (programs) that are available in software markets.

The first actor, named **Ar1**, is a data management system with a screen form interface.

Actor **Ar1** has four functions:

1. **F1**: insert data
   - inputs:
     - \( I_{11} \): a command (statement) that let Ar1 start working;
     - \( I_{12} \): datum to be inserted into a database
   - outputs:
functions

INPUTS / OUTPUTS

I11: a command
I12: datum to be inserted
O11: datum inserted
O12: a signal showing the end of the execution
I21: a command
I22: datum to be deleted
O21: a signal showing the end of the execution
O22: a number indicating how many data deleted

I31: a command
I32: datum to be modified
O31: datum modified
O32: a signal showing the end of the execution
I41: a command
I42: datum to be fetched
O41: datum fetched
O42: a signal showing the end of the execution

Figure 6.13: Actor Ar₁: a data management system
- \( O_{11} \): a signal showing that \texttt{Ar1} has finished the execution;
- \( O_{12} \): datum inserted into the database.

- **function:**
The function inserts a set of data into a database, in either an interactive way through a screen form or a batch processing way.

2. **F2**: delete data

- **inputs:**
  - \( I_{21} \): a command (statement) that let \texttt{Ar1} start working;
  - \( I_{22} \): a datum deleted from a database;

- **outputs:**
  - \( O_{21} \): a signal showing that \texttt{Ar1} has finished the execution;
  - \( O_{22} \): a number informing how many data have been deleted.

- **function:**
The function deletes a set of data from a database.

3. **F3**: modify data

- **inputs:**
  - \( I_{31} \): a command (statement) that let \texttt{Ar1} start working;
  - \( I_{32} \): a datum is to be modified;

- **outputs:**
  - \( O_{31} \): a signal showing that \texttt{Ar1} has finished the execution;
  - \( O_{32} \): a datum modified and sent back to the database.

- **function:**
The function modifies a set of data in database, either interactively through a screen form or by batch processing.

4. **F4**: query

- **inputs:**
  - \( I_{41} \): a command (statement) that let \texttt{Ar1} start working;
  - \( I_{42} \): a datum fetched from a database.

- **outputs:**
  - \( O_{41} \): a signal showing that \texttt{Ar1} has finished the execution;
  - \( O_{42} \): a datum sent out by the function.

- **function:**
The function fetches a set of data from a database, either interactively through a screen form or by batch processing.

The second actor, named \texttt{Ar2}, is a report generator. This actor has been used as an example in chapter 3 (see Figure 3.36) and is reused in this chapter (see Figure 6.13 with minor differences.

**Ar2** has two functions:
6.4. The actor model

Figure 6.14: Actor \( A_{r2} \): a report generator

1. **F1**: compile report specification
   - **input**: 
     - \( I_{11} \): a report specification.
   - **outputs**: 
     - \( O_{11} \): a generated report format;
     - \( O_{12} \): the error message for a report specification.
   - **function**: The function compiles a report specification, then generate a compiled format of the report, or gives an error message when the specification is not correct.

2. **F2**: generate report
   - **inputs**: 
     - \( I_{21} \): a report format;
     - \( I_{22} \): a set of data to be put on the report;
     - \( I_{23} \): a printer’s name.
   - **outputs**: 
     - \( O_{21} \): a generated report.
   - **function**: The function generates a report from a set of data, according to a pre-defined format. It prints the report on a specified printer if the printer name is given, otherwise it prints the report on a default printer.
6.4.2 Mapping the actors to reusable processes

The two actors are both translated to become reusable processes when the mapping rules in section 3.4.3 are applied. The two processes are drawn in Figure 6.15 and Figure 6.16 respectively (by two kinds of graphical notations for the processes), where the renamed inputs/outputs and I/O conditions are all provided with the descriptions of \textbf{Ar1} and \textbf{Ar2}.
Figure 6.17: The cross references between scenarios and data stores
Figure 6.18: The cross references between some processes and actors that have been mapped into general purpose processes
6.5 The cross references

The cross references among the components of different sub-models are given in Figure 6.17 and Figure 6.18. Not all cross references have been illustrated, since we have only decomposed some parts of the system functions (P1, P2 and P3 of the function programming). However, because other processes are to be supported by the same technical systems, we believe that the given descriptions have shown quite well how the sub-models are integrated within our modeling framework.

In Figure 6.17 we can find the relationships between the data stores and the external scenarios. In section 4.2 we know the participation manners of the classes Externalscenario and store are partial_1 and partial_n respectively. This is reflected in the example: each data store references only one scenario while a scenario may be referenced by several data stores (e.g. the scenario Confirmation_payment_registration), and some data stores may not reference any scenario of the data model (e.g., S3.1.1 in Figure 6.11).

Figure 6.18 specifies how the two actors are used to support some processes in the system activities. Actor Ar1 is used to insert data into data stores and query data from data stores; Ar2 is used to produce formatted reports of some data by fetching data from a data store and using a pre-defined format that is also saved in a data store.

6.6 Brief summary

An example of information processing activities in an organization has been studied in detail in this chapter. The example shows why we need to refine the concept “information system” into concepts at two levels: “business system” and “technical system”, and shows how the market available products of software can be used to support the information processing activities.

What have we learnt from the case study? First we claim that the example provides some convincing arguments supporting the motivation and the problem solutions of the proposed modeling approach. The arguments have been given in previous parts of the thesis. However, only with this practical example – though small but quite complete – can we claim that a better approach has been developed, or in other words, we have made some progress to improve modeling techniques in the field of information systems. Now we list several points as a brief summary of the case study:

- the information management in an organization is a set of activities involving people, technical systems and data to be managed. In order give clear descriptions of the complex processes and objects, a modeling
method must be able to illustrate different aspects separately as well as be able to link these aspects together. In other words, we need an integrated system model with several sub-models. The example shows that our modeling approach has met the requirement;

- when more formal properties are specified, e.g., the port constructs attached to processes, we can get a better description and correspondingly better understanding of a system;

- We need to define some processes whose process logic are impossible to know (e.g. some human being’s activities such as process P3.2) or unnecessary to know (e.g. the processes supported by actor Ar1 and Ar2, see Figure 6.18). This example provides a solution in the extreme: no process logic is needed, since we just use market available software products. Only when we want to automate more parts of the system we need to specify the process logics of these parts and build new actors according to the logics. Anyhow, we are not creating information systems but are updating parts of the information systems that we always have in our social life.
Chapter 7

Concluding remarks

In the preface of this thesis we illustrated some problems of modern information system modeling techniques. It is clear that these problems should be solved, particularly because of the rapid development of computer & network technologies, and the corresponding increasing requirement to develop more computer-aided information systems faster. The question is how to improve our modeling methods and tools so that they can help both IT professionals and common people to build their systems that are needed.

The answer in the thesis is that: first we suggest a set of criteria for a “good” modeling, based on the ontological study in Chapter 1 on the concept of information system, then we propose the integrated modeling framework COMIS which includes an extended E-R model, a formal process model and an actor model and connects the components in different sub-models by means of cross references. Another part of the modeling framework is a set of methods for consistency verification or checking, based on both theoretic and practical considerations.

To what degree have we have reached the objectives that we formulated at the beginning of the thesis? In order to show the result of our work, we have done a case study in Chapter 6, which reveals that the new modeling approach has provided some of the desirable features specified in Chapter 1. Now we summarize the results more explicitly, including achievements as well as weak points that need to be improved. Then we mention some work to be done in the future.

7.1 Comments on the modeling approach

In order to let the evaluation of our modeling approach be comparable with the comments given to the existing modeling methods, we apply the same set
of criteria:

- Perspectives. COMIS has provided full perspectives on all the aspects we claimed to be considered:
  - the data perspective is supported with an extended E-R model ONE-R which can be used to specify complex data types and customized operations on the data objects;
  - the process perspective is supported with the Process Port Model (PPM) that is an extension of the DFD model with more formal concepts appended;
  - the event/behavior perspective is supported with the auxiliary concept triggering rules in PPM. By using the rules, we may specify when and under what condition an agent sends the triggering message to invoke a process.
  - the role perspective is supported by the Actor Model (AM) and its cross references with PPM, i.e., how the actors can be used as a general purpose and reusable process in the PNDs and how the bottom-level processes can be grouped and leads to the generation of an actor.

- Application scope and abstraction. One of the basic motivations is to let a modelling method enable the specification of the information management activities in a larger scope, rather than only around a single technical system. Therefore, the concept information system is defined at two levels: the business systems and the technical systems. On the business system level, we support a multi-leveled decompositions to specify the system activities in a step-wise way, then at the bottom level, we can link the processes with the functions of technical systems.

- Formal specifications. COMIS has provided formal specifications for ONE-R (the internal type descriptions) and PPM and AM (the auxiliary concepts for ports and flows. On this basis, we have developed the formal methods and algorithms for checking the consistency of the system specifications.

- User-friendly descriptions. ONE-R is user-friendly, since it is a semantic data model with an SQL-like language whereas all the formal descriptions are hidden from the users. PPM and AM are user-friendly too, because the original DFD model is very user-friendly and we just append some auxiliary concepts that seem not too difficult to the users.

Having shown that the COMIS framework does provide the desirable properties for a good modeling method, according to the criteria we suggested in Chapter 1, we can claim that our approach has made some improvement to the existing modeling techniques. However, we would also like to indicate some weak points of our approach:
7.2. Further work

- We call COMIS a "modeling framework" because we have not fully developed a specification language. In some other approaches, e.g., the object-oriented approach, a specification language has been defined in great detail [117] whereas in our approach only the data model has been given a well-defined language. In other parts only the concepts are provided, though it is not impossible to define a more complete specification language to integrate the conceptual components more tightly;

- The consistency check methods is built in a more restricted scale than that of the modeling framework. For instance, in ONE-R we allow definition of methods on the entity and relationship classes, but we did not consider the methods when transforming the types into a logical theory, since we only provide static verification to the data schemas. On the other hand, the triggering rules have not been treated as the objects to be checked, although the importance of the logical correctness of the rules is never in question. Besides, our approach may specially expose its limitation on the systems with strong time dimension requirements, because the concept of event has not been given a more important role in the COMIS framework.

- The complexity in calculating an STD for a process network diagram may be a serious problem. No doubt it is an exponential problem and may easily be too complex to be calculated, if we allow the random structure of PNDs.

7.2 Further work

The purpose of the work presented in this thesis is to build a more solid foundation for the development of information systems, especially for the development of CASE tools. Developments of modern ISs can probably not be done without CASE tools; on the other hand, a CASE tool can be more powerful only when it is based on a better modeling method. The work reported in this thesis was done during the development of the CASE tool PPP. During that project, many problems were exposed that motivated the work reported in this thesis.

The major work to be done in the future is to develop a new PPP that will absorb the results of our modeling approach. Not only the conceptual components, but also the consistency check methods must be realized within the new system.

However, every time when we find a solution to solve some problem, we will perhaps find other problems. Some problems with my work has been indicated, and we must do more work to solve them. This work will include:

- Developing a more complete specification language. At present only for ONE-R a language with well-defined syntax has been developed. For
other parts of COMIS, the languages should also be developed so that COMIS has a complete specification language. With such a language, a system model, even though it may be expressed by various graphical notations, can still be fully described by a collection of statements. This will help to specify a system more accurately, and help the work for formal consistency check.

- ONE-R, as a pure conceptual model, should be able to specify the types of data stored in different databases with different DBMSs, thus various mapping methods should be further developed in order to let ONE-R work practically on different technical platforms. On the other hand, the concept of scenario should be further elaborated to let ONE-R be used for modeling of the inter related multi-databases (also called federated databases);

- Making more efforts to develop the logical verifications that can check the correctness of operations (methods) defined on classes in ONE-R and the triggering rules in PPM;

- Define some “normal forms” of the PNDs, which adapt some special topological structures that may be proven to be efficient to reduce the complexity of the calculation for constructing STDs and be helpful to guide users to obtain more reasonable system process structures.
Appendix A

The SDL Syntax

In this section the main parts of the SDL syntax is given. Some non-terminal symbols such as various names, "num(ber)" , and "program-code" are not further specified, since they are identical with those in SQL or they depend on the implementation environment.

During the process of designing SDL, two versions of SQL: the first complete representation of SQL [32] and the SQL standard [147], were referenced, thus many elements from them are absorbed by the SDL syntax. However, SDL is still a language which is conceptually different from SQL, since the underlying data model of SDL is ONE-R, an extension and variant of the E-R model, rather than the relational data model which is the basis of SQL.

<sdl statement> ::= <ddl statement> | <query> | <dml statement>
                 | <control statement> | <cursor statement>

<ddl statement> ::= <build-scenario statement>
                 | <build-external-scenario statement>
                 | <merge-scenario statement> | <modify-scenario statement>
                 | <remove-scenario statement>

<build-scenario statement> ::= BUILD SCENARIO <scenario-name>
                             [ {<scenario-element>;};... ] END ;

<scenario-element> ::= <date-type definition>
                    | <entity-class definition> | <relationship-class definition>
                    | <method definition> | <constraint definition>

<data-type definition> ::= DATA TYPE <data-type-name> = <type-spec>

<type-spec> ::= <rename constructor> | <tuple constructor>
              | <set constructor> | <type-union constructor>

213
<rename constructor> ::= ( <data-type> )

<data-type> ::= <data-type-name> | INTEGER | REAL | BOOLEAN
  | STRING[[<num>]]

<tuple constructor> ::= ( [attribute-name][*]: <data-type-name>
  [{, attribute-name: <data-type}>...] )

<set constructor> ::= '{' <data-type-name> '}'[[<num> : <num>]]

<type-union constructor> ::= U( [attribute-name: <data-type>
  [{, attribute-name: <data-type}>...] )

<entity-class definition> ::= ENTITY CLASS <class-name> =
  <entity-class-spec>

<entity-class-spec> ::= '{' [[IS_A_SUBCLASS_OF : <class-name>;]]
  <class-body> '}'

<class-body> ::= <attributes-spec> [method-spec]

<attribute-spec> ::= ATTRIBUTES: [attribute-name][*]:
  <data-type1>; [{[attribute-name][*]: <data-type1>;}]...

<data-type1> ::= [UNIQUE] [PRIVATE] <data-type>

<method-spec> ::= METHODS: <method-name>; [{<method-name>;}...]

<relationship-class definition> ::=?
  RELATIONSHIP CLASS <class-name> = <relationship-class-spec>

<relationship-class spec> ::= '{' <relationship-spec> <class-body> '}'

<relationship-spec> ::= ON_ENTITY_CLASSES : <class-name>
  [{, <class-name>}]...: <rel-type>

<rel-type> ::= <rel-type1> <rel-type2>

<rel-type1> ::= ( [map-num] [:<map-num>]... )

(map-num) ::= 1 | n

<rel-type2> ::= ( [par-manner] [:<par-manner>]... )

<par-manner> ::= f | p

<method definition> ::= METHOD <method-spec>

<method-spec> ::= <method-name>([parameter-spec]): <data-type>
  <parameter-type-spec> '{' <method-body> '}'
<parameter-spec> ::= <parameter-name> [{, <parameter-name>}...] 
<parameter-type-spec> ::= <parameter-type> [{, <parameter-type>}...] 
<parameter-type> ::= <parameter-name> ::= <data-type> [,PRIVATE]; 
<method-body> ::= <program-code> 
<constraint-definition> ::= CONSTRAINT <constraint-name>: 
<constraint-body> 
<constraint-body> ::= <entity-class-list> <cons-spec> | <boolean> 
<entity-class-list> ::= <entity-class-name> [{, <entity-class-name}>...] 
<cons-spec> ::= ARE DISTINCT 
| [MAKE_UP | PARTITION] <entity-class-name> 
<build-external-scenario statement> ::= 
   BUILD EXTERNAL SCENARIO <scenario-name> 
   <reference-spec> [{<ex-clause> ;}... END ; 
<reference-spec> ::= REFERENCING <scenario-name> [{, <scenario-name>}...] 
<ex-clause> ::= <accessible-clause> | <alias-clause> 
| <derive-clause> 
<accessible-clause> ::= ACCESSIBLE CLASSES: <accessible classes> 
<accessible classes> ::= <class-name> [{, <class-name>}...] 
<alias-clause> ::= ALLIASES: <alias-def> [{, <alias-def>}...] 
<alias-def> ::= <alias-name> = <class-name> 
<derive-clause> ::= DERIVED CLASSES: <derive-def> [{; <derive-def}>...] 
<derive-def> ::= <class-name> = <query-expr> 
<merge-scenario statement> ::= MERGE SCENARIOS <scenarios> INTO 
   <scenario-name> ; 
<scenarios> ::= <scenario-name> [{, <scenario-name>}...] 
<modify-scenario statement> ::= MODIFY SCENARIO <scenario-name> 
   [.modify-clause...] END ; 
<modify-clause> ::= <drop-clause> | <scenario-element> | <ex-clause>
<drop-clause> ::= DROP <element-name> ;

<remove-scenario statement> ::= REMOVE SCENARIO <scenario-name> ;

<query> ::= <query-expr> [<order-clause>]

<order-clause> ::= ORDER BY <order-spec-list>

<query-expr> ::= <query-block> | <query-expr> <set-op> <query-block>

<set-op> ::= INTERSECT | UNION | MINUS

<query-block> ::= SELECT [DISTINCT] <sel-expr-list> [INTO <target-list>]
FROM <from-list> [WHERE <boolean>]
[GROUP BY <group-spec-list> [HAVING <boolean>]]

<sel-expr-list> ::= { | ALL} | <sel-expr> [(, <sel-expr>)...]

<sel-expr> ::= <var-name>.{ | ALL} | [<attri-name> = ] <expr>

<expr> ::= <arith-term> | <expr> <add-op> <arith-term>

<add-op> ::= + | -

<arith-term> ::= <arith-factor> | <arith-term> <multi-op> <arith-factor>

<arith-factor> ::= [<add-op>] <primary>

<primary> ::= <attri-spec> [<further-projection>]
| set-fn [DISTINCT] ( <expr> ) | ( <expr> ) | COUNT (*)
| <constant> | <method-name> (<para-list>)

<set-fn> ::= AVG | MAX | MIN | SUM | COUNT

<attri-spec> ::= <attri-name> | <var-name>.<attri-name>
| <attri-spec>.<attri-name>

<further-projection> ::= ( <project-list> )
| '{' <var-name> '}'< ( <project-list> ) [WHERE <boolean>]
| U ( <project_list> ) [{, ( <project_list> })...]

<project-list> ::= <project-item> [{, <project-item}>...]

<project-item> ::= [ [<var-name> =] <expr>

<para-list> ::= <expr> [{, <expr}>...]

<target-list> ::= <host-location> [{, <host-location}>...]

<from-list> ::= <from-item> [{, <from-item}>...]
<from-item> ::= <variables> IN <class-name>

<variables> ::= <var-name> [{, <var-name>}]...

<boolean> ::= <boolean-clause> | IF <boolean-clause> THEN <boolean-clause>
             | { FORALL | EXISTS } <var-name> (boolen-clause)

<boolean-clause> ::= <boolean-term> | <boolean-clause> OR <boolean-term>

<boolean-term> ::= <boolean-factor> | <boolean-term> AND <boolean-factor>

<boolean-factor> ::= [NOT] <boolean-primary>

<boolean-primary> ::= <predicate> | (<boolean>)

<predicate> ::= <expr> <comparison> <expr>
             | <attri-spec> '('<var-name>')'
             | <variables> {RELATES|RELATE} <variables> THROUGH <class-name>
             | <expr> <comparison> <sub-query>
             | <expr> [NOT] BETWEEN <expr> AND <expr>
             | <expr> [NOT] IN {<expr> | <sub-query>}
             | <expr> [NOT] IN <literal>
             | <var-name> [NOT] IN <class-name>
             | (<sub-query>) CONTAINS {ALL|SOME} (<sub-query>)
             | EXISTS <sub-query>

<comparison> ::= = | > | < | >= | <= | ≠ | CONTAINS

<sub-query> ::= SELECT [DISTINCT] <sel-expr-list>
              FROM <from-list> [WHERE <boolean>]
              [GROUP BY <group-spec> [HAVING <boolean>]]

<literal> ::= '{<lit-tuple-list>}' | <lit-tuple> | <constant>

<lit-tuple-list> ::= <lit-tuple> [{, <lit-tuple>}]...

<constant> ::= <string> | num | <host-location> | NULL
             | '{<constant> [{, <constant>}]}'

<lit-tuple> ::= (<entry-list>)

<entry-list> ::= <entry> [{, <entry>}]...

<entry> ::= [<attri-name>:] <constant> | [<attri-name>:] <lit-tuple>
          | <literal>

<group-spec-list> ::= <group-spec> [{, <group-spec>}]...

<group-spec> ::= <attri-spec> | <class-name>

<order-spec-list> ::= <attri-spec> [{, <attri-spec}>]...
<dml statement> ::= <insertion> | <deletion> | <update> | <add-rel> | <remove-rel>

<insertion> ::= INSERT INTO <receiver> : <insert-spec>

<receiver> ::= <class-name> [<attributes>]

<attributes> ::= <attri-name> [{, <attri-name}>...]

<insert-spec> ::= VALUES <lit-tuple> [<and-also-clause>...]

<and-also-clause> ::= AND_ALSO (<insertion>) [<in-relationship>]

<in-relationship> ::= IN_RELATIONSHIP <class-name> WITH <varibles>
   IN <class-names> WHERE <boolean>
   | IN_RELATIONSHIP <class-name> WITH LAST_TUPLE

<class-names> ::= <class-name> [{, <class-name}>...]

<deletion> ::= DELETE <var-name> FROM <class-name> [WHERE <boolean>]

<update> ::= UPDATE <var-name> IN <class-name> <set-value-spec>
   [WHERE {CURRENT <cursor-name> | <boolean>}]}

<set-value-spec> ::= SET VALUE WITH <method-name>(<para-list>)
   | SET <set-clause-list>

<set-clause-list> ::= <set-clause> [{, <set-clause}>...]

<set-clause> ::= <attri-name>[<further-projection>] = <expr>

<add-rel> ::= ADD RELATIONSHIP TO <class-name> FOR <varibles>
   IN <class-names> WHERE <boolean>

<remove-rel> ::= REMOVE RELATIONSHIP FROM <class-name> FOR
   <varibles> IN <class-names> WHERE <boolean>

<control statement> ::= <grant> | <revoke> | BEGIN TRANSACTION
   | END TRANSACTION

<grant> ::= GRANT {READ ONLY ACCESS | FULL ACCESS} ON
   <scenario-name> TO <user-list>

<user-list> ::= <user-name> [{, <user-name}>...]

<revoke> ::= REVOKE [{READ ONLY ACCESS | FULL ACCESS}] ON
   <scenario-name> FROM <user-list>

<cursor statement> ::= <declare> | <open> | <fetch> | <close>
\[<\text{declare}> ::= \text{DECLARE} \ <\text{cursor-name}> \ \text{CURSOR} \ \text{FOR} <\text{query}>\]

\[<\text{open}> ::= \text{OPEN} <\text{cursor-name}>\]

\[<\text{fetch}> ::= \text{FETCH} <\text{cursor-name}> [\text{INTO} <\text{target-list}>]\]

\[<\text{close}> ::= \text{CLOSE} <\text{cursor-name}>\]
Appendix B

The proofs for theorem 3.1, 3.2 and 3.3

In this appendix section we present the proofs for the theorems given in section 3.3.4. The proofs are provided in a relatively simple way:

- we just list the non-logical rules (the formulae of the theory), and call those logical theorems that can be deduced from a generic first order axiomatic system logical rules;
- when context is clear, we just get a new formula directly and omit some detailed steps.

The Universe of Discourse \( D \): The set of all port names;

The constants and variables \( P_1, P_2, \ldots \);

The Function Themes:

\[ \text{and}(P_1, \ldots, P_n): D \times \ldots \times D \rightarrow D \ (n \geq 1); \]

\[ \text{xor}(P_1, \ldots, P_n): D \times \ldots \times D \rightarrow D \ (n \geq 1); \]

The Predicates \( \text{Filled}(P) \) and \( \text{NotUsed}(P) \);

The rules (formulae) of the theory:

**R-I.** \( \forall P_1, \ldots, \forall P_n \ \text{Filled}(\text{and}(P_1, \ldots, P_n)) \leftrightarrow \text{Filled}(P_1) \land \ldots \land \text{Filled}(P_n) \)

**R-II.** \( \forall P_1, \ldots, \forall P_n \ \text{Filled}(\text{xor}(P_1, \ldots, P_n)) \leftrightarrow (\text{Filled}(P_1) \land \text{NotUsed}(P_2) \land \ldots \land \text{NotUsed}(P_n)) \land \
\lor (\text{NotUsed}(P_1) \land \text{Filled}(P_2) \land \ldots \land \text{NotUsed}(P_n)) \land \
\lor \ldots \lor (\text{NotUsed}(P_1) \land \text{NotUsed}(P_2) \land \ldots \land \text{Filled}(P_n)) \)
Appendix B. The proofs for theorem 3.1, 3.2 and 3.3

R-III. \( \forall P_1, \ldots, \forall P_n \text{NotUsed}(and(P_1, \ldots, P_n)) \iff \text{NotUsed}(P_1) \lor \cdots \lor \text{NotUsed}(P_n) \)

R-IV. \( \forall P_1, \ldots, \forall P_n \text{NotUsed}(xor(P_1, \ldots, P_n)) \iff \text{NotUsed}(P_1) \land \cdots \land \text{NotUsed}(P_n) \)

R-V. \( \forall P_1, \ldots, \forall P_n \text{Filled}(P) \land \text{NotUsed}(P) \Rightarrow \text{NIL} \)

R-VI. \( \forall P_1, \ldots, \forall P_n \text{Filled}(P) \lor \text{NotUsed}(P) \)

The definition for specifying that two ports \( P_1 \) and \( P_2 \) are equivalent:

\[ P_1 \text{ equals } P_2 \equiv_{def} \text{Filled}(P_1) \iff \text{Filled}(P_2) \]

From the theory, we can conduct following theorems:

**Theorem 3.1:**

\[ \forall P_{11}, \ldots, P_{1m}, \forall P_2, \ldots, \forall P_n \text{ and } (\text{and}(P_{11}, \ldots, P_{1m}), P_2, \ldots, P_n) \]

\[ \text{equals} \]

\[ (P_{11}, \ldots, P_{1m}, P_2, \ldots, P_n). \]

**proof.** According to the definition for port equivalence, We should prove that

\[ \forall P_{11}, \ldots, P_{1m}, \forall P_2, \ldots, \forall P_n \ (\text{Filled}(\text{and}(P_{11}, \ldots, P_{1m}), P_2, \ldots, P_n)) \]

\[ \iff \text{Filled}(\text{and}(P_{11}, \ldots, P_{1m}, P_2, \ldots, P_n)) \]

We first list the inference steps for

\[ \text{Filled}(\text{and}(P_{11}, \ldots, P_{1m}, P_2, \ldots, P_n)) \]

\[ \Rightarrow \text{Filled}(\text{and}(P_{11}, \ldots, P_{1m}, P_2, \ldots, P_n)) \]

1. \( \text{Filled}(\text{and}(P_{11}, \ldots, P_{1m}, P_2, \ldots, P_n)) \) (assumption)
2. \( \text{Filled}(\text{and}(P_{11}, \ldots, P_{1m})) \land \text{Filled}(P_2), \ldots \land \text{Filled}(P_n) \) (R-I, logical rules)
3. \( \text{Filled}(P_{11}) \land \cdots \land \text{Filled}(P_{1m}) \land \cdots \land \text{Filled}(P_n) \) (R-I, logical rules)
4. \( \text{Filled}(\text{and}(P_{11}, \ldots, P_m, P_2, P_n)) \) (R-I, logical rules)

The formula

\[ \text{Filled}(\text{and}(P_{11}, \ldots, P_{1m}, P_2, \ldots, P_n)) \Rightarrow \]

\[ \text{Filled}(\text{and}(P_{11}, \ldots, P_{1m}), P_2, \ldots, P_n)) \]

can be proven with an inverse inference sequence. With the logical generalization rule, the theorem is then proven.

**Theorem 3.2:**
\[ \forall P_{11}, \cdots, \forall P_{1m}, \forall P_2, \cdots, \forall P_n \ xor(P_{11}, \cdots, P_{1m}, P_2, \cdots, P_n) \]

equals
\[ xor(P_{11}, \cdots, P_{1m}, P_2, \cdots, P_n). \]

**proof.** According to the definition for port equivalence, we should prove that
\[ \forall P_{11}, \cdots, \forall P_{1m}, \forall P_2, \cdots, \forall P_n \ (Filled(xor(P_{11}, \cdots, P_{1m}), P_2, \cdots, P_n)) \]
\[ \iff Filled(xor(P_{11}, \cdots, P_{1m}, P_2, \cdots, P_n))) \]

We first list the inference steps for
\[ Filled(xor(P_{11}, \cdots, P_{1m}), P_2, \cdots, P_n)) \]
\[ \Rightarrow Filled(xor(P_{11}, \cdots, P_{1m}, P_2, \cdots, P_n)) \]

(1) \( Filled(xor(P_{11}, \cdots, P_{1m}), P_2, \cdots, P_n)) \) (assumption)

(2) \((Filled(xor(P_{11}, \cdots, P_{1m})) \land \text{NotUsed}(P_2)) \cdots \land \text{NotUsed}(P_n)) \)
\[ \lor(\text{NotUsed}(xor(P_{11}, \cdots, P_{1m})) \land \text{Filled}(P_2) \cdots \land \text{Filled}(P_n)) \]
\[ \ldots \]
\[ \lor(\text{NotUsed}(xor(P_{11}, \cdots, P_{1m})) \land \text{NotUsed}(P_2) \cdots \land \text{Filled}(P_n)) \]
\[ (\text{R-II, logical rules}) \]

(3) \(((Filled(P_{11}) \land \text{NotUsed}(P_{12}) \cdots \land \text{NotUsed}(P_{1m})) \)
\[ \lor(\text{NotUsed}(P_{11}) \land \text{Filled}(P_{12}) \cdots \land \text{NotUsed}(P_{1m})) \]
\[ \ldots \]
\[ \lor(\text{NotUsed}(P_{11}) \land \text{NotUsed}(P_{12}) \cdots \land \text{Filled}(P_{1m})) \]
\[ \land(\text{NotUsed}(P_2) \cdots \land \text{NotUsed}(P_n)) \)

\[ \lor((\text{NotUsed}(P_{11}) \land \cdots \land \text{NotUsed}(P_{1m})) \]
\[ \land\text{Filled}(P_2) \cdots \land \text{NotUsed}(P_n)) \]
\[ \ldots \]
\[ \lor((\text{NotUsed}(P_{11}) \land \cdots \land \text{NotUsed}(P_{1m})) \]
\[ \land\text{NotUsed}(P_2) \cdots \land \text{Filled}(P_n)) \]
\[ (\text{R-II, logical rules}) \]

(4) \((Filled(P_{11}) \land \text{NotUsed}(P_{12}) \cdots \land (P_{1m}) \land \text{NotUsed}(P_2) \cdots \land \text{NotUsed}(P_n)) \)
\[ \lor(\text{NotUsed}(P_{11}) \land \text{Filled}(P_{12}) \cdots \land (P_{1m}) \land \text{NotUsed}(P_2) \cdots \land \text{NotUsed}(P_n)) \]
\[ \ldots \]
\[ \lor(\text{NotUsed}(P_{11}) \land \text{NotUsed}(P_{12}) \cdots \land (P_{1m}) \land \text{NotUsed}(P_2) \cdots \land \text{Filled}(P_n)) \]
\[ (\text{logical rules}) \]

(5) \( Filled(xor(P_{11}, \cdots, P_{1m}, P_2, \cdots, P_n)) \) (R-II)

The formula

\[ Filled(xor(P_{11}, \cdots, P_{1m}, P_2, \cdots, P_n)) \Rightarrow \]
\[ Filled(xor(xor(P_{11}, \cdots, P_{1m}), P_2, \cdots, P_n)) \]
can be proven with an inversed inference sequence. With the logical generalization rule, the theorem is then proven.

**Theorem 3.3:**

\[
\forall P_{11}, \cdots, \forall P_{1m}, \forall P_2, \cdots, \forall P_n \text{ and } (\text{xor}(P_{11}, \cdots, P_{1m}), P_2, \cdots P_n)
\]

\[\text{equals}\]

\[\text{xor}(\text{filled}(P_{11}, P_2, \cdots, P_n), \cdots \text{and}(P_{1m}, P_2, \cdots, P_n)).\]

**proof.** According to the definitio for port equivalence, We should prove that

\[\forall P_{11}, \cdots, \forall P_{1m}, \forall P_2, \cdots, \forall P_n \text{ Filled}(\text{xor}(P_{11}, \cdots, P_{1m}), P_2, \cdots P_n)\]

\[\Leftrightarrow \text{filled}(\text{xor}(P_{11}, P_2, \cdots P_n), \cdots \text{and}(P_{1m}, P_2, \cdots P_n)).\]

We first list the inference steps for

\[\text{Filled}(\text{xor}(P_{11}, \cdots, P_{1m}), P_2, \cdots P_n))\]

\[\Rightarrow \text{Filled}(\text{xor}(P_{11}, P_2, \cdots P_n), \cdots \text{and}(P_{1m}, P_2, \cdots P_n)).\]

(1) \text{Filled}(\text{xor}(P_{11}, \cdots, P_{1m}), P_2, \cdots P_n)) \ (assumption)

(2) \text{Filled}(\text{xor}(P_{11}, \cdots, P_{1m})) \land \text{Filled}(P_2), \cdots \land \text{Filled}(P_n) \ (R-I)

(3) (\text{Filled}(P_{11}) \land \cdots \land \text{NotUsed}(P_{1m}))

\[
\begin{align*}
\ldots \\
\text{v}(\text{NotUsed}(P_{11}) \land \cdots \land \text{Filled}(P_{1m})) \\
\land (\text{Filled}(P_2) \land \cdots \land \text{Filled}(P_n)) \ (R-II, \text{logical rules}) \\
\text{d}) \\
\text{v}(\text{Filled}(P_{11}) \land \text{Filled}(P_2) \land \cdots \land \text{Filled}(P_n) \land \text{NotUsed}(P_{12}) \land \cdots \land \text{NotUsed}(P_{1m}))
\end{align*}
\]

(4) (\text{Filled}(P_{11}) \land \text{Filled}(P_2) \land \cdots \land \text{Filled}(P_n) \land \text{NotUsed}(P_{12}) \land \cdots \land \text{NotUsed}(P_{1m}))

\[
\begin{align*}
\ldots \\
\text{v}(\text{Filled}(P_{11}) \land \text{Filled}(P_2) \land \cdots \land \text{Filled}(P_n) \land \text{NotUsed}(P_{11}) \land \cdots \land \text{NotUsed}(P_{1m-1})) \\
\land (\text{NotUsed}(P_{12}) \lor \text{NotUsed}(P_2) \lor \cdots \lor \text{NotUsed}(P_n)) \ (R-I, \text{logical rules}) \\
\text{d) } \\
\text{v}(\text{Filled}(P_{11}) \land \text{Filled}(P_2) \land \cdots \land \text{Filled}(P_n) \land \text{NotUsed}(P_{12}) \land \cdots \land \text{NotUsed}(P_{1m}))
\end{align*}
\]

(5) (\text{Filled}(P_{11}, P_2, \cdots, P_n)) \land \text{NotUsed}(P_{12}) \land \cdots \land \text{NotUsed}(P_{1m})

\[
\begin{align*}
\ldots \\
\text{v}(\text{Filled}(P_{11}, P_2, \cdots, P_n)) \land \text{NotUsed}(P_{12}) \land \cdots \land \text{NotUsed}(P_{1m}) \\
\land (\text{NotUsed}(P_{1m}) \lor \text{NotUsed}(P_2) \lor \cdots \lor \text{NotUsed}(P_n)) \ (R-I, \text{logical rules}) \\
\text{d) } \\
\text{v}(\text{Filled}(P_{11}, P_2, \cdots, P_n)) \land \text{NotUsed}(P_{11}) \lor \text{NotUsed}(P_2) \lor \cdots \lor \text{NotUsed}(P_n) \\
\land \cdots \\
\land (\text{NotUsed}(P_{1m}) \lor \text{NotUsed}(P_2) \lor \cdots \lor \text{NotUsed}(P_n))
\end{align*}
\]
\[
\begin{align*}
\text{Filled}(\text{and}(P_{1m}, P_2, \ldots, P_n)) \\
\land (\text{NotUsed}(P_{11}) \lor \text{NotUsed}(P_2) \cdots \lor \text{NotUsed}(P_n)) \\
\land \cdots \\
\land (\text{NotUsed}(P_{1m-1}) \lor \text{NotUsed}(P_2) \cdots \lor \text{NotUsed}(P_n)) \\
) \quad \text{(logical rules)} \\
(7) \quad \text{Filled}(\text{and}(P_{11}, P_2, \ldots, P_n)) \land \text{NotUsed}(\text{and}(P_{12}, P_2, \ldots, P_n)) \\
\land \cdots \land \text{NotUsed}(\text{and}(P_{1m}, P_2, \ldots, P_n)) \\
) \\
\text{Filled}(\text{and}(P_{12}, P_2, \ldots, P_n)) \land \text{NotUsed}(\text{and}(P_{11}, P_2, \ldots, P_n)) \\
\land \cdots \land \text{NotUsed}(\text{and}(P_{1m}, P_2, \ldots, P_n)) \\
) \\
\text{Filled}(\text{and}(P_{1m}, P_2, \ldots, P_n)) \land \text{NotUsed}(\text{and}(P_{11}, P_2, \ldots, P_n)) \\
\land \cdots \land \text{NotUsed}(\text{and}(P_{1m-1}, P_2, \ldots, P_n)) \\
) \quad \text{(R-III, logical rules)} \\
(8) \quad \text{Filled}(\text{xor}(\text{and}(P_{11}, P_2, \ldots, P_n), \ldots, \text{and}(P_{1m}, P_2, \ldots, P_n))) \quad \text{(R-II)}
\end{align*}
\]

In the reverse direction, we also list the inference steps for

\[
\text{Filled}(\text{xor}(\text{and}(P_{11}, P_2, \ldots, P_n), \ldots, \text{and}(P_{1m}, P_2, \ldots, P_n))) \\
\Rightarrow \text{Filled}(\text{and}(\text{xor}(P_{11}, \cdots, P_{1m}), P_2, \ldots, P_n))
\]

1. \text{Filled}(\text{xor}(\text{and}(P_{11}, P_2, \ldots, P_n), \ldots, \text{and}(P_{1m}, P_2, \ldots, P_n))) \quad \text{(assumption)}
2. \text{Filled}(\text{and}(P_{11}, P_2, \ldots, P_n)) \land \text{NotUsed}(\text{and}(P_{12}, P_2, \ldots, P_n)) \cdots \\
\land \text{NotUsed}(\text{and}(P_{1m}, P_2, \ldots, P_n)) \\
\text{Filled}(\text{and}(P_{12}, P_2, \ldots, P_n)) \land \text{NotUsed}(\text{and}(P_{11}, P_2, \ldots, P_n)) \cdots \\
\land \text{NotUsed}(\text{and}(P_{1m}, P_2, \ldots, P_n)) \\
\quad \cdots \\
\text{Filled}(\text{and}(P_{1m}, P_2, \ldots, P_n)) \land \text{NotUsed}(\text{and}(P_{11}, P_2, \ldots, P_n)) \cdots \\
\land \text{NotUsed}(\text{and}(P_{1m-1}, P_2, \ldots, P_n)) \quad \text{(R-II, logical rules)}
3. \quad ((\text{Filled}(P_{11}) \land \text{Filled}(P_2) \cdots \land \text{Filled}(P_n)) \\
\land (\text{NotUsed}(P_{12}) \lor \text{NotUsed}(P_2) \cdots \text{NotUsed}(P_n)) \\
\cdots \\
\land (\text{NotUsed}(P_{1m}) \lor \text{NotUsed}(P_2) \cdots \text{NotUsed}(P_n)) \\
) \\
\text{Filled}(P_{12}) \land \text{Filled}(P_2) \cdots \land \text{Filled}(P_n) \\
\land (\text{NotUsed}(P_{11}) \lor \text{NotUsed}(P_2) \cdots \text{NotUsed}(P_n)) \\
\cdots \\
\land (\text{NotUsed}(P_{1m}) \lor \text{NotUsed}(P_2) \cdots \text{NotUsed}(P_n)) \\
) \\
\text{Filled}(P_{1m}) \land \text{Filled}(P_2) \cdots \land \text{Filled}(P_n) \\
\land (\text{NotUsed}(P_{11}) \lor \text{NotUsed}(P_2) \cdots \text{NotUsed}(P_n)) \\
\quad \cdots
\[ \land (\text{NotUsed}(P_{1m-1}) \lor \text{NotUsed}(P_2) \cdots \text{NotUsed}(P_n)) \]
\]
\[ ) \text{ (R-I, R-II, logical rules)} \]
\[ (4) (\text{Filled}(P_{11}) \land \text{Filled}(P_2) \cdots \land \text{Filled}(P_n) \land \text{NotUsed}(P_{12}) \cdots \land \text{NotUsed}(P_{1m})) \]
\[ \lor (\text{Filled}(P_{12}) \land \text{Filled}(P_2) \cdots \land \text{Filled}(P_n) \land \text{NotUsed}(P_{11}) \cdots \land \text{NotUsed}(P_{1m})) \]
\[ \ldots \]
\[ \lor (\text{Filled}(P_{1m}) \land \text{Filled}(P_2) \cdots \land \text{Filled}(P_n) \land \text{NotUsed}(P_{11}) \cdots \land \text{NotUsed}(P_{1m-1})) \]
\[ ) \text{ (logical rules)} \]
\[ (5) (\text{Filled}(P_2) \land \cdots \land \text{Filled}(P_n)) \land \]
\[ (\text{Filled}(P_{11}) \land \text{NotUsed}(P_{12}) \cdots \land \text{NotUsed}(P_{1m})) \]
\[ \lor (\text{Filled}(P_{12}) \land \text{NotUsed}(P_{11}) \cdots \land \text{NotUsed}(P_{1m})) \]
\[ \ldots \]
\[ \lor (\text{Filled}(P_{1m}) \land \text{NotUsed}(P_{11}) \cdots \land \text{NotUsed}(P_{1m-1})) \]
\[ ) \text{ (logical rules)} \]
\[ (6) \text{Filled}(\text{xor}(P_{11}, \cdots , P_{1m})) \land \text{Filled}(P_2) \cdots \land \text{Filled}(P_n) \text{ (R-II, logical rules)} \]
\[ (7) \text{Filled}(\text{and}(\text{xor}(P_{11}, \cdots , P_{1m}), P_2, \cdots , P_n)) \text{ (R-I)} \]

On the basis of the inference steps, by applying the generalization rule, the theorem is then proven.
List of Figures

1.1 The system as a "Black Box" ........................................ 8
1.2 A "reaction" IS ...................................................... 10
1.3 Application systems on a DB ..................................... 10
1.4 Technical systems in an organization ............................ 12
1.5 The system concepts in the FRISCO approach ................. 18

2.1 The leveled DFD structure ........................................ 25
2.2 Schema of World Traveler database ............................... 29
2.3 The structure diagrams for two entities ......................... 32
2.4 The system as a simulation of the real world .................. 33
2.5 The system model in SSADM ....................................... 35
2.6 The functional decomposition in IE .............................. 38
2.7 System models built with BNM and REMORA ................... 41

3.1 A business system model .......................................... 52
3.2 A technical systems model .................................... 53
3.3 An integrated information system modeling framework ......... 54
3.4 The ONE-R modeling structure ................................ 58
3.5 The Entity and Relationship Classes in IFIP_conference ........ 59
3.6 The data types in ONE-R ........................................ 60
3.7 Methods with their inputs and output ........................... 61
3.8 The classes, data types and methods in the data model for the IFIP Conference example .......................... 62
3.9 Mapping the reality into the computer world .................. 63
3.10 The ONE-R type constructors ................................ 70
3.11 The data types in the IFIP example ............................. 70
3.12 The internal forms of the entity and relationship types in the IFIP example ................................ 71
3.13 The internal forms of the entity and relationship classes in the IFIP example ................................ 72
3.14 The four-leveled type hierarchy in ONE-R ..................... 73
3.15 The result type of the product operator ....................... 74
3.16 The result type of a project operation ......................... 76
3.17 The result type of a nest operation ............................ 77
3.18 The result type of a packed relationship join operation .... 80
3.19 The three leveled schema structure of SDL ................... 84
3.20 Comparing ONE-R with other data models on some aspects ... 88
3.21 Basic components of PPM ...................................... 91
3.22 A process $P$ and its i/o condition ............................ 93
3.23 The activities for ordering tickets in the IFIP conference .......... 94
3.24 Two port structures with the same flow combinations .......... 96
3.25 Comparison of an or port with and ports .................. 96
3.26 Equivalent port trees ................................ 99
3.27 The equivalent port structure for a process $P$ ............. 100
3.28 Data flows consisting of flow-pipe and input/output ........... 101
3.29 The canonical PNDs of IFIP ticket booking activities .......... 103
3.30 Reuse of a process and the graphical representations .......... 104
3.31 Reuse of a general purpose process .......................... 105
3.32 A reusable process and its augmented reference ................ 106
3.33 Two flowing manners from data stores ..................... 107
3.34 Cross references between PPM and ONE-R ..................... 108
3.35 The is-a hierarchy of object types in COMIS .................... 110
3.36 The model of an actor .................................. 111
3.37 Mapping the actor into a reusable process (the process is drawn
by two kinds of graphical notations) ......................... 113
3.38 Grouping processes into an actor ............................ 113

4.1 The meta model of COMIS (1) .................................. 125
4.2 The meta model of COMIS (2) .................................. 126
4.3 A compact unifiability digraph ................................. 131
4.4 The i/o condition for the process network for $P_1$ of the IFIP
ticket booking activities .................................... 136
4.5 A decomposition of a process $P_1$ ................................ 138
4.6 Two ways for consistency check on the decomposition of $P_1$ .... 139
4.7 Two process networks which may produce run-time errors ........ 140
4.8 Conducting possible execution sequences of a process .......... 142
4.9 The execution life cycle and the operation groups of a process . 146
4.10 The STD of the process network for $P_1$ in the IFIP ticket book-
ing activities .................................................. 151
4.11 The canonical port structure and the STD of the process network .... 152
4.12 The canonical port structure and the partially feasible STD of
a process network ................................................ 153
4.13 An STD and all its paths ...................................... 154
4.14 Construction of the port structures through the STD in Figure 13156

5.1 The overall structure of the PPP tool .............................. 160
5.2 The overall structure of the PPP Modeling Framework .......... 161
5.3 The PPP window for its modeling framework ...................... 162
5.4 The PM window for drawing, checking and validation ............ 163
5.5 The PrM window for drawing, checking and validating .......... 164
5.6 A process is decomposed into a new PrM diagram ................ 165

6.1 Scenario$_1$: the entity classes and relationship classes processed
by the program committee ....................................... 175
6.2 Scenario$_2$: the entity classes and relationship classes processed
by the organizing committee ..................................... 177
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>PND <em>programming.d₀</em>: the processes executed in the program committee</td>
<td>181</td>
</tr>
<tr>
<td>6.4</td>
<td>More information about the components in <em>programming.d₀</em></td>
<td>182</td>
</tr>
<tr>
<td>6.5</td>
<td>PND <em>organizing.d₀</em>: the processes executed in the organizing committee</td>
<td>186</td>
</tr>
<tr>
<td>6.6</td>
<td>More information about the components in <em>organizing.d₀</em></td>
<td>187</td>
</tr>
<tr>
<td>6.7</td>
<td>The decomposition for <strong>P1</strong></td>
<td>191</td>
</tr>
<tr>
<td>6.8</td>
<td>More information for the decomposition of <strong>P1</strong></td>
<td>192</td>
</tr>
<tr>
<td>6.9</td>
<td>The decomposition for <strong>P2</strong></td>
<td>194</td>
</tr>
<tr>
<td>6.10</td>
<td>More information about the decomposition of <strong>P2</strong></td>
<td>195</td>
</tr>
<tr>
<td>6.11</td>
<td>The decomposition for <strong>P3</strong></td>
<td>198</td>
</tr>
<tr>
<td>6.12</td>
<td>More information for the decomposition of <strong>P3</strong></td>
<td>199</td>
</tr>
<tr>
<td>6.13</td>
<td>Actor <strong>Ar₁</strong>: a data management system</td>
<td>201</td>
</tr>
<tr>
<td>6.14</td>
<td>Actor <strong>Ar₂</strong>: a report generator</td>
<td>203</td>
</tr>
<tr>
<td>6.15</td>
<td>Mapping actor <strong>Ar₁</strong> to a reusable process</td>
<td>204</td>
</tr>
<tr>
<td>6.16</td>
<td>Mapping actor <strong>Ar₂</strong> to a reusable process</td>
<td>204</td>
</tr>
<tr>
<td>6.17</td>
<td>The cross references between scenarios and data stores</td>
<td>205</td>
</tr>
<tr>
<td>6.18</td>
<td>The cross references between some processes and actors that have been mapped into general purpose processes</td>
<td>206</td>
</tr>
</tbody>
</table>
Bibliography


BIBLIOGRAPHY


