Shaoying Liu

Formal Engineering for Industrial Software Development

Using the SOFL Method

Springer

Formal Engineering
for Industrial Software Development

Springer-Verlag Berlin Heidelberg GmbH

Shaoying Liu

Formal Engineering
for Industrial
Software Development

Using the SOFL Method

With 90 Figures and 30 Tables

Springer

Shaoying Liu
Department of Computer Science
Hosei University
Tokyo, 184-8584
Japan
sliu@k.hosei.ac.jp

To my family

Foreword

In any serious engineering discipline, it would be unthinkable to construct a large system without a precise notion of what is to be built. Equally, any professional engineer must record not only his or her proposed solution to an engineering challenge, but also reasons why the solution is believed to be correct. Software engineering faces the challenge of creating very large systems and must therefore solve both of these challenges. Combined with established good practice such as inspections, *formal methods* can make a significant impact on software dependability.

The fact that descriptions and correctness arguments were required was obvious to pioneers of computing as early as von Neumann and Turing, who both wrote about ways of reasoning about programs. Since their early attempts, the need has been to find *tractable ways* of coping with systems of ever increasing size. The landmark contributions of Bob Floyd and Peter Naur culminated in Tony Hoare's wonderfully clear exposition of "axioms" for reasoning about programming constructs. This in turn led to development methods like VDM, Z, and B. Such methods work well for systems which are sequential and self-contained, but extensions were required to deal with other real world problems such as concurrency and "open" systems where obtaining specifications (and recognising that the requirements will evolve over the lifetime of the system) is as challenging as developing the "closed" components which result.

This book brings together ideas from VDM and from object-oriented thinking to propose an approach to the development of realistic software systems. "SOFL" builds on some of the most pervasive ideas to come from theoretical computing science and amalgamates them into an approach which the author has used on a variety of practical applications. Such books are to be whole-heartedly welcomed because they are written with an acute understanding of the issues for designers of useful software.

The success and pervasiveness of object-oriented methods suggest that it is unnecessary to say more about their marriage with formal methods since it might appear to be an obvious step. I should however like to add some arguments in favour of this specific combination. It is frequently argued that

today's computer applications are inherently complex. I think only part of this complexity is inevitable in today's systems. Of course, the code for an online airline seat reservation system of 2003 is bound to be larger than the code for a simple batch payroll system of the 1960s. But it is also clear that much of today's software is very poorly structured: its architecture is often opaque and users find it almost impossible to form a mental model of how it works. With a WYSIWYG word processor, this can result in frustration and expensive loss of productivity for professional users; for safety-critical applications, poorly understood systems present the real danger of an operator making life threatening mistakes. The ultimate contribution of formal methods will be to help clean up the architecture of systems, and the marriage with object-oriented ideas is important in this regard.

Another key contribution of object-oriented implementations is that they offer a way of controlling interference in concurrent computing. Interference is the key characteristic of concurrent programs (whether the parallel programs share states or interact only by communication primitives). Reasoning about interference can be delicate and complex; good engineers will reduce the areas where such complexity is required to a minimum. Object-oriented implementations put the control of interference where it belongs: that is, with the designer.

The combination of formalism and object-oriented design has the potential to yield clean and accurate implementations. The reader is encouraged to understand and use SOFL.

Cliff B. Jones
University of Newcastle upon Tyne

Preface

This book aims to give a systematic introduction to SOFL (*S*tructured *O*bject-Oriented *F*ormal *L*anguage) as one of the *Formal Engineering Methods* for industrial software development. Formal engineering methods are a further development of formal methods toward industrial application. They support the integration of formal methods into the software development process, the construction of formal specifications in a user-friendly manner, and rigorous but practical verification of software systems. SOFL achieves all of these features by integrating data flow diagrams, Petri nets, VDM, and the object-oriented approach in a coherent manner for specifications construction, and by integrating formal verification with fault tree analysis and testing for reviewing and testing specifications. It also provides a way to transform formal specifications into Java programs. SOFL has been taught for many years at universities, and has also been applied to systems modelling and design both in industry and academia.

Formal methods have made significant contributions to the establishment of theoretical foundations and rigorous approaches for software development over the last 30 years. They emphasize the use of mathematical notation in writing system specifications, both functional and non-functional, and the employment of formal proofs based on logical calculus for verifying designs and programs. However, despite a few exceptions, most formal methods have met challenges lobbying for acceptance by industrial users. A lack of appropriate education may be seen as one of the major reasons for this unfortunate situation, but, apart from this, a bigger problem is that formal methods have not successfully addressed many important engineering issues related to their application in industrial environments. For example, how can formal specifications, especially for large-scale systems, be written so that they can be easily read, understood, modified, verified, validated, and transformed into designs and programs? How can the use of formal, semi-formal, and informal methods be balanced in a coherent manner to achieve the best quality assurance under practical schedule and cost constraints? How can formal proof and testing, static analysis, and prototyping techniques be combined to achieve rigorous

and effective approaches to the verification and validation of formal specifi-
cations, designs, and programs? How can the refinement from unexecutable
formal specifications into executable programs be effectively supported? How
can the evolution of specifications at various levels be assisted and controlled
consistently and efficiently? How can software development processes be for-
mally managed so that they can be well predicated before they are carried
out, and well controlled during their implementations? And how can effec-
tive software tools supporting the use of formal methods be built so that the
productivity and reliability of systems can be enhanced?

Since the research to provide possible solutions to these questions addresses
a different aspect of the problem; I call this area *Formal Engineering Methods.*
In other words, formal methods emphasize the utilization of mathematical no-
tation and calculus in software development, without considering the human
factor (e.g., capability, skills, educational background) and other uncertainties
(e.g., accuracy and completeness of requirements, changes in both specifica-
tions and programs, the scale and complexity of systems), whereas formal en-
gineering methods advocate the incorporation of mathematical notation into
the software engineering process to substantially improve the rigor, compre-
hensibility, and effectiveness of commonly used methods for the development
of real systems in the industrial setting.

After introducing the general ideas of formal engineering methods, this
book provides a tutorial on the recently developed formal engineering method
SOFL. The material originally evolved from my research publications over last
15 years, from courses, and from seminars offered at universities and compa-
nies in Japan, UK, USA, and Australia. It is intended to be the basis for
courses on formal engineering methods, but it also contains the latest new
research results in the field. By reading through this book, the reader will
find that SOFL has provided many useful ideas and techniques as solutions
to many of the questions raised above. It not only makes formal methods
accessible to engineers, but also makes the use of formal methods enjoyable
and effective. In order to help readers study SOFL easily, I have tried to
make the descriptions as precise and comprehensible as possible. I have also
tried to avoid unnecessary formal semantics of SOFL constructs, to the ex-
tent that this does not affect our understanding them. Numerous examples
are given throughout the book to help the explanation of the SOFL specifi-
cation language and method, and many exercises are prepared for readers to
improve their understanding of the material they have studied and to check
their progress.

The objective of this book is to bring readers to the point where they can
use SOFL to construct specifications by evolving informal specifications to
semi-formal ones, and then to formal ones. It is also intended to help readers
to master rigorous and practical techniques for verifying and validating speci-
fications, to learn the process of developing software systems using SOFL, and
to get new ideas for building intelligent software engineering environments.

Audience

This book is written for people who want to improve their knowledge and skills in developing complex software systems. Readers who are interested in formal methods, but frustrated by using them in practice, will benefit greatly from this book. Although I have made efforts to make the book as self-contained as possible, and have provided many exercises for individual study, the reader will need some experience in programming and basic knowledge of discrete mathematics to appreciate and digest some of the abstract material.

Using This Book

This book can be used at the second year undergraduate or above level as a computer science textbook for courses on *logic and formal specification*, *advanced software engineering*, and *software specification, verification, and validation*, respectively. According to my experience at Hosei University and other institutions, in the course on *logic and formal specification* that takes about 24 hours, the fundamental knowledge on first order logic and skills for writing comprehensible formal specifications for large-scale software systems can be introduced based on the contents of chapters 1 to 12.

The course on *advanced software engineering* usually takes 26 hours, incorporating rigorous software development techniques using a formal specification language, including skills for writing modular, hierarchical, and comprehensible formal specifications, evolving informal specifications to semi-formal and then to formal ones, transforming structured abstract design into an object-oriented detailed design, and transforming detailed design into object-oriented programs in Java. The contents of this course can contain chapters 1, 4 to 16, 19, and 20.

In the course on *software specification, verification, and validation*, which is suitable for graduate students and needs about 24 hours, the techniques for writing formal specifications and for their verification and validation can be introduced based on the contents of chapters 4 to 18.

The book can also be used as a reference book to support the study of other related courses or individual study of formal engineering methods for software development. To make the book easier to use, I have organized the materials into nine parts:

Introduction. Chapter 1 explains the motivation of formal engineering methods and describes what they are. After discussing the problems in software engineering and difficulties in using formal methods, I describe the general ideas and features of formal engineering methods and their relation with SOFL.

Logic. Chapters 2 and 3 introduce mathematical logic that is adopted by SOFL. Both propositional logic and predicate logic are explained, and their application to the writing of and reasoning about SOFL specifications are discussed.

Specification. Chapters 4 to 6 cover the most important components of SOFL specifications: *module*, *hierarchy of modules*, and *explicit specifications*.

We explain the techniques of combining graphical notation and formal textual notation in writing comprehensible but formal specifications with these components.

Data types. Chapters 7 to 12 describe all the built-in data types in SOFL, which include basic types, set types, sequence and string types, composite and product types, map types, and union types. Each type is introduced by explaining its constructors and operators, and their use in specifications.

Classes. Chapter 13 is concerned with the concept of class: a user-defined data type. We discuss the structure of classes by explaining their similarity with and differences from modules, and the way to use classes in module specifications.

Software process. Chapters 14 and 15 present a software development process using SOFL from informal specifications to programs, and in particular elaborate several techniques for constructing formal specifications in an evolutionary manner.

Case study. Chapter 16 describes a case study of specifying an ATM (Automated Teller Machine) using the SOFL specification language. This case study is designed to show the entire process of developing a detailed design specification from an informal user requirements specification, and gives the reader an opportunity to review and digest the contents studied before this chapter.

Verification and validation. Chapters 17 and 18 introduce two techniques for verification and validation of specifications: rigorous reviews and specification testing. We explain how formal proof and the practical techniques like reviews and testing are integrated to provide rigorous but practical methods for verification and validation of specifications.

Transformation and software tools. Chapter 19 explains the principle and technique for the transformation of design specifications into Java programs, including data transformation and functional transformation; the last chapter, 20, discusses the potential features of an intelligent software engineering environment supporting formal engineering methods, in particular SOFL, and its importance in enhancing the productivity and reliability of software products.

All readers are recommended to read Chapter 1, but those who are experienced in programming and have sufficient knowledge about mathematical logic can skip Chapters 2 and 3. Chapters 4 to 6 present the fundamental principles and techniques for constructing specifications, and therefore are suitable for all readers. Chapters 7 to 12, concerned with abstract data types, need attention from the beginners, but can be quickly browsed by those who are familiar with VDM (Vienna Development Method), with caution because of the differences in syntax. Chapters 13 to 20 contain specific materials on SOFL and are recommended for study by all readers.

in 1989 at the University of Manchester in the UK, was motivated by Cliff
B. Jones's book titled "Systematic Software Development using VDM" (first
edition), and benefited from the seminars and discussions he provided for the
formal methods group while I was studying for my PhD in Manchester. I am
grateful to John Latham for his constructive comments on the initial work
on the integration of VDM and Data Flow Diagrams, which establishes the
foundation for the development of SOFL. The initial integration work also
benefited from Tom DeMarco's book titled "Structured Analysis and System
Specification" and from my experience of working with John A. McDermid at
the University of York. My sincere thanks also go to the people whose joint
work with me has impacted on the development of both the SOFL language
and the method presented in this book. Chris Ho-Stuart defined an operational
semantics for SOFL, and provided many suggestions on the improvement of
the SOFL language. Jeff Offutt developed an approach to testing programs
based on SOFL specifications. Yong Sun worked out with his research student
a prototype of a graphical user interface (GUI) for SOFL. Jin Song Dong pro-
vided a denotational semantics for SOFL using Object-Z. My former students
Tetsuo Fukuzaki and Koji Miyamoto developed a prototype specification test-
ing tool and a GUI for SOFL, respectively. I would also like to express my
gratitude to all the research partners and my students who have completely or
partially applied SOFL to develop their software systems, or combined SOFL
with other methods for software development. I appreciate very much the
feedback from my students after they read the draft of the book. Financial
support from the Ministry of Education, Culture, Sports, Science and Tech-
nology of Japan through several research grants is gratefully acknowledged.
Finally, my thanks go to three anonymous referees for their constructive com-
ments and suggestions, and the editor Ralf Gerstner of Springer-Verlag for his
encouragement and suggestions that helped me to improve the initial draft
and for his painstaking efforts in the editing of the text.

Contents

1

Introduction

The development of complex software systems on a large scale is usually a complicated activity and process. It may involve many developers, possibly with different backgrounds, who need to work together as a team or teams in order to ensure the productivity and quality of systems within a required schedule and budget. Each developer plays a specific role, for example, as an analyst, designer, programmer, or tester, and is usually required to produce necessary documents. The documents may need to be provided to other developers in the team for reading or for assisting them in performing their tasks. For this reason the documents need to be well presented, with appropriate languages or notations, so that they can be understood accurately and used effectively.

In the early days of computing, software was seen as synonym of program, but this view was gradually changed after the birth of the field *software engineering* in the late 1960s [1, 2]. Software is no longer regarded only as a program, but as a combination of documentation and program. In other words, documentation is part of software that represents different aspects of the software system. For example, the documentation may contain the user's requirements, the goal to be achieved by a program, the design of the program, or the manual for using the program.

The documentation is important for ensuring the quality and for facilitating maintenance of a program system. If the documentation containing the user's requirements or the program design is difficult to understand accurately by the developers undertaking subsequent development tasks, the risk of producing an unsatisfactory program system will run high. The consequence of this can be serious: the program system either needs more time and effort to be improved to the level that is deliverable or needs to be completely rebuilt. In either case, a loss of money and time is unavoidable.
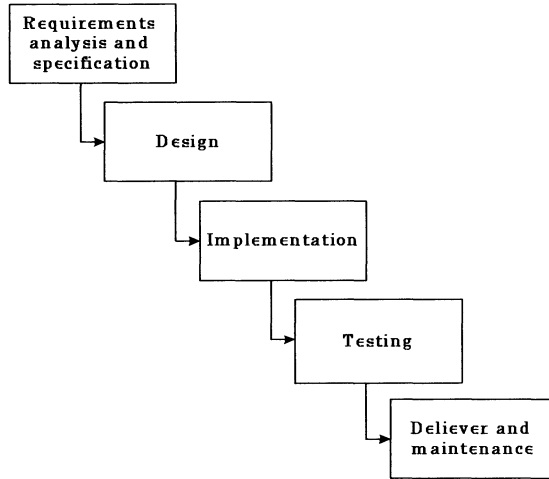
Fig. 1.1. The waterfall model for software development

1.1 Software Life Cycle

Software, like a human being, has a *life cycle*, composed of several phases. Each of these phases results in the development of either a part of the system or something associated with the system, such as a specification or a test plan [32]. A typical software life cycle, known as *waterfall model*, is given in Figure 1.1. Although the real picture of the software life cycle may be much more complicated than the waterfall model, it depicts primary features of the software development process. Almost every other model uses the idea of the waterfall model as its foundation [11, 84, 8, 111, 98].

The typical waterfall life cycle model comprises five phases: *requirements analysis and specification*, *design*, *implementation*, *testing*, and *delivery and maintenance*.

Requirements analysis and specification is a study aiming to discover and document the exact requirements for the software system to be constructed [23][51][52]. To this end, the system in the real world, which is to be computerized, may need modeling so that all the necessary requirements can be explored. The result of such a study is usually a document that defines the identified requirements. A requirement in the document can be a statement, a formal logical expression, a text, a diagram, or their combinations that tell *what is to be done* by the system. Such a document is usually called a *requirements specification*. For example, "build a student information system" can be an abstract level requirement.

Design is an activity to construct a system, at a high level, to meet the system requirements. In other words, design is concerned with *how* to provide a solution for the problem reflected in the requirements [56]. For this reason, design is usually carried out on the basis of the requirements specification.

Design can be done in two stages: *abstract design* and *detailed design*. Abstract design is intended to build the architecture of the entire system that defines the relation between software modules or components. Detailed design usually focuses on the definition of data structures and the construction of algorithms [15, 99]. The result of design is a document that represents the abstract design and detailed design. Such a document is called *design* or *design specification*. To distinguish between the activity of design and the document resulting from the design activity, we use *design* to mean the design activity and *design specification* to mean the design document in this book.

Implementation is where the design specification is transformed into a program written in a specific programming language, such as Pascal [37], C [58], or Java [4]. The implemented program is executable on a computer where the compiler or interpreter of the programming language is available. The primary concerns in implementation are the functional correctness of the program against its design and requirements specifications.

Testing is a way to detect potential faults in the program by running the program with test cases. As there are many ways to introduce faults during the software development process, detecting and removing faults are necessary. Testing usually includes the three steps: (1) test case generation; (2) the execution of the program with the test cases; and (3) test results analysis [115, 53].

There are two approaches to program testing: *functional testing* and *structural testing*, which are distinguished by their purposes and the way test cases are generated. Functional testing, also known as *black-box testing*, aims to discover faults leading to the violation of the consistency between the specification and the program, and test cases are generated based on the functional specification (requirements specification or design specification or both) [45, 9, 108]. Structural testing, alternatively known as *white-box testing*, tries to examine every possible aspect of the program structure to discover the faults introduced during the implementation, and test cases are therefore generated based on the program structure [106]. In general, both functional testing and structural testing are necessary for testing a program system because they are complementary in finding faults.

Deliver and maintenance is where the system is ultimately delivered to the customer for operation, and is modified either to fix the existing faults when they occur during operation or to meet new requirements [111]. Maintenance of a system usually requires a thorough understanding of the system by maintenance engineers. To enhance the reliability and efficiency of maintenance, well documented requirements specification and design specification are important and helpful.

In addition to the forward flow from upper level phases to lower level phases in the software life cycle, we should also pay attention to the backward flow from lower level phases to upper level phases. Such a backward flow represents a feedback of information or verification. For example, it is desirable to check whether the design specification is consistent with the requirements

specification, whether the implementation satisfies the design specification, and so on.

1.2 The Problem

One of the primary problems in software projects is that the requirements documented in specifications may not be accurately and easily understood by the developers carrying out different tasks. The analyst may not understand correctly and completely the user requirements due to poor communication; the designer may misunderstand some functional requirements in the specification due to their ambiguous definitions; the programmer may make a guess of the meaning of some graphical symbols in the design specification; and so on. The major reason for this problem is the use of informal or semi-formal language or notation, such as natural language (e.g., English) and diagrams that lack a precise semantics. Let us consider the requirements for a *Hotel Reservation System* as an example:

A Hotel Reservation System manages information about rooms, reservations, customers, and customer billing. The system provides the services for making reservations, checking in, and checking out. A customer may make reservations, change, or cancel reservations.

This specification defines necessary resources to be managed and desirable operations to be provided for the management of the resources. The resources include rooms, reservations, customers, and customer billing. The operations are making reservation, checking in, checking out, changing reservations, and canceling reservations. As all the terms representing either resources or operations are given in English, they may be interpreted differently by different developers. For instance, by customers the analyst might mean persons with a full name, address, telephone, and room reservations, but the programmer may misunderstand it as persons with only a full name; by checking in the analyst might mean that the customer has arrived at the hotel, obtained the room key, and made payment for all his or her room charges in advance, but the programmer may misunderstand that checking in does not require advanced payment.

This problem is caused not only by the lack of the detailed and precise definition of the terms, but also by the *free* style of the documentation. Informal specifications can be written in a manner where every important term is defined in detail, but the free style of writing may make the specification tedious and keep important information hidden among irrelevant details. In fact, a well-organized documentation, even if written in an informal language or notation, can greatly help improve its readability. However, no matter how much the organization is improved in an informal documentation, it is usually impossible to guarantee no misunderstanding occurs because ambiguity

Formal Methods =
Formal Specification
+
Formal Verification
———————————
↑   ↑
Set theory, logic,
algebra, etc.
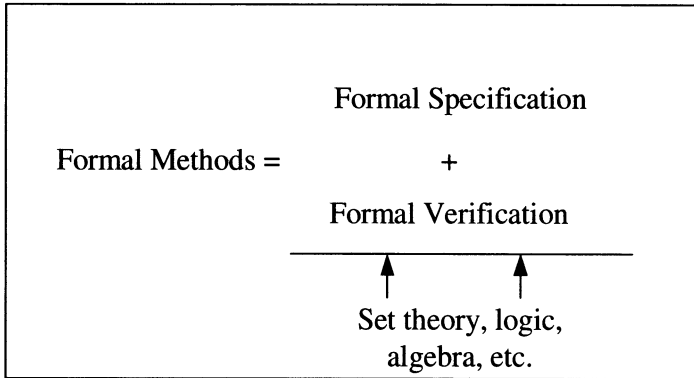
Fig. 1.2. The description of formal methods

is an intrinsic feature of informal languages. Furthermore, in an informal description it is difficult to show the clear relations among different parts of a complicated specification.

A specification should be *consistent* in defining requirements, that is, no contradiction should exist between different requirements in the specification. The specification is also expected to document all the possible user requirements; such a property is called *completeness* of specification. Since informal specifications lack formality in both syntax and semantics, it is usually difficult, even impossible in most cases, to support automated verification of their consistency and completeness in depth. Furthermore, informal specifications offer no firm foundation for design and coding, and for verifying the correctness of implemented programs in general.

1.3 Formal Methods

One way to improve the quality of documentation and therefore the quality of software is to provide formalism in documentation. Such a formalized documentation offers a precise specification of requirements and a firm basis for design and its verification.

1.3.1 What Are Formal Methods

Formal methods for developing computer systems embrace two techniques: *formal specification* and *formal verification* [55, 3, 38, 116, 43]. Both are established based on elementary mathematics, such as set theory, logic, and algebraic theory, as illustrated in Figure 1.2.

Formal specification is a way to abstract the most important information away from irrelevant implementation detail and to offer an unambiguous documentation telling *what is to be done* by the system. A formal specification is
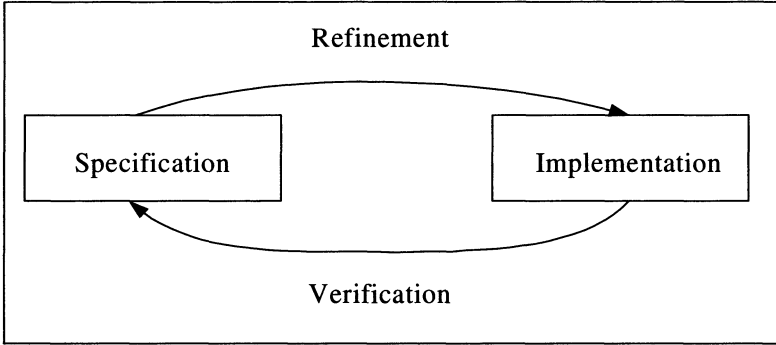
Fig. 1.3. The principle of formal methods

written in a language with formal syntax and semantics. Of course, programming languages are also formal languages, but they are for implementation of computer systems, not for specifications. In a specification language, there is usually a mechanism that allows the definition of what to be done by the system without the need of giving algorithmic solutions, whereas in a programming language all the mechanisms are usually designed for writing algorithmic solutions (i.e., code). For this reason, formal specifications are more concise and abstract than programs.

Formal verification is a way to *prove* the *correctness* of programs against their specifications [42][24][36][101]. A program is correct if it does exactly what the specification requires. The proof of the correctness is usually based on a logical calculus that provides necessary axioms and inference rules. An axiom is a statement of a fact without any hypothesis, while a rule is a statement of a fact under some hypotheses. Program correctness proof aims to establish a logical consistency between the program and its specification.

A *method* offers a way to do something. This is true to formal methods as well. Figure 1.3 shows the principle of formal methods. A specification is constructed first, and then refined into a program by following appropriate refinement rules [90][6]. In general, since this refinement may not be done automatically, the correctness of the program may not be ensured. Therefore, a formal verification of the program against its specification is needed to ensure its correctness. Such a verification may sometimes also help detect faults in the specification.

In principle, the activities of *specification*, *refinement*, and *verification* advocated by formal methods may not necessarily be completed within a single cycle; they are usually applied repeatedly to several level specifications. Thus, an entire software development can be modeled as a successive refinement and verification process, after the informal requirements are formalized into the highest level formal specification and the specification is validated against
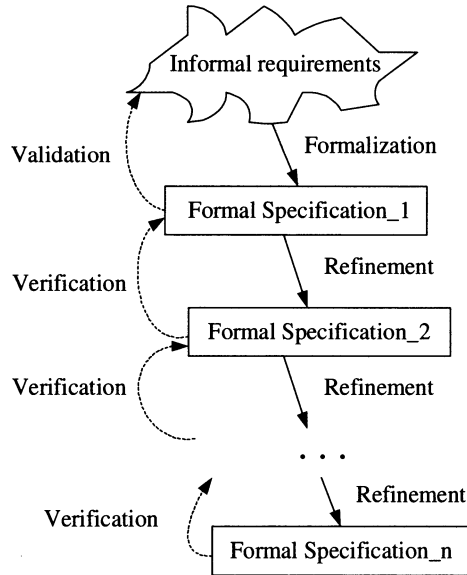
Fig. 1.4. Software development using formal methods

the informal requirements, as illustrated in Figure 1.4. In this model, each level document is perceived as a specification of the next lower level document, and each refinement takes the current level specification more toward the final executable program, represented by the lowest level specification (i.e., formal specification_n). Since refinement is a transitive relation between specifications, the final program must theoretically satisfy the highest level specification (i.e., formal specification_1).

1.3.2 Some Commonly Used Formal Methods

Many formal methods have been reported in the literature so far, such as VDM, Z, B-Method, HOL [35], PVS [20], Larch [39], RAISE [38], and OBJ [34, 31], but in accordance with the international survey on industrial applications of formal methods [19] and the applications described in Hinchey and Bowen's edited book [88], the most commonly used formal mthods include VDM, Z, and B-Method.

VDM (The Vienna Development Method) offers a notation, known as VDM-SL (VDM-Specification Language), and techniques for modelling and designing computing systems. It was originally developed based on the work of the IBM Vienna Laboratory in the middle 1970s. The publication of Jones's book titled "Systematic Software Development using VDM" [54, 55] has contributed considerably to the wide spread of VDM technology in education and application. The most important feature in VDM is the mechanism for defining operations. An operation can be regarded as abstraction of a procedure

in the programming language Pascal (or similar structure in other programming languages) and defined with a precondition and a postcondition. The precondition imposes a constraint on the initial state before operation, while the postcondition presents a constraint on the final state after operation. The most essential technique in writing an operation specification is definition of a relation between the initial and final states in the postcondition of the operation. This technique allows the specification to focus on the description of the function of the operation, and therefore facilitates the clarification of functional requirements before providing them with a program solution. In order to model complex systems, VDM provides a set of built-in types, such as *set*, *sequence*, *map*, and *composite* types. In each type, necessary *constructors* and *operators* are defined, which allow for the formation and manipulation of objects (or values) of the type. Using those built-in types as well as their constructors and operators in specifications, complex functions of operations can be modeled precisely and concisely. With the progress in software supporting tools [29], VDM has been gradually adopted in the development of industrial systems, and has been extended to VDM++ to support object-oriented design [27].

Z was originally designed as a formal notation based on axiomatic set theory and first order predicate logic for describing and modeling computing systems by the Programming Research Group at Oxford University around 1980 [107][100], and later developed to a method by providing rules for refinement and verification [116]. An essential component used in Z specifications is known as *schema*. A schema is a structure that can be used to define either system state or operation. The definition of state includes the declarations of variables and their constraints given as predicate expressions. A schema defining an operation is usually composed of two parts: declarations and predicates. The declarations may include declarations of input variables and/or output variables, as well as the related state schemas. The predicates impose constraints on the input variables, output variables, and the related state variables. Complex specifications can be formed by using the *schema calculus* available in the Z notation. Although Z uses syntax different from VDM, they share the similar model-based approach to writing formal specifications. Based on Z notation, other formal notations have also been developed to support object-oriented design and concurrency, such as Object-Z [105] and TCOZ [81].

The B-method has been developed by Jean-Raymond Abrial. It provides an Abstract Machine Notation for writing system specifications and rules for refinement of specifications into programs [3, 102]. A specification in B is constructed by means of defining a set of related abstract machines. An abstract machine is similar to a module in VDM, which contains local state variables, invariants on the state, and necessary operations. Each machine must have a name in order to allow other machines in a large specification to refer to it. A machine can extend another machine in order to expand its contents (e.g., state and operations) and include another machine in order to allow for calling

of its operations. Following the refinement rules, an abstract specification can be transformed step by step into a concrete representation (or implementation in B termnology) that can then be translated into a program of a specific programming language. With the progress of tools development, the B-Method has been applied in a few industrial projects [44].

1.3.3 Challenges to Formal Methods

In my opinion formal methods have presented the most reasonable, rigorous, and controllable approach to software development so far, at least theoretically, but their application requires high skills in mathematical abstraction and proof. The situation seems that if all the suggested steps in formal methods could be taken in practice, with no compromise, we would have no doubt in the correctness of the program produced. However, since software engineering is a human activity (with support of software tools), the effect of formal methods depends heavily on whether and how they can be applied in practice by software engineers, usually with many constraints. The major challenges are:

- Formal specifications for large-scale software systems are usually more difficult to read than informal specifications, and this would be aggravated for complex systems. Informal specifications are usually easy to read, but offer no guarantee of correct understanding because of ambiguity in language semantics. Formal methods offer precise specifications, but they are difficult to read, and there is no guarantee of correct understanding either. The two cases may result in the similar situation that the reader of the specification would make a guess about the meaning of some expressions, but for different reasons. The specification may be too *imprecise* to be correctly understood in the first case whereas it may be too *difficult* to be correctly understood in the second case.
- Formal verification of program correctness is too expensive to be deployed in practice. Although it is the most powerful technique for demonstrating the consistency between programs and their specifications among existing verification techniques, such as testing, static analysis, animation, and model checking, but only a small number of experts can apply this technique, and it may not be cost-effective for complex systems. Except for safety-critical systems or the safety-critical parts of systems, formal verification is usually out of reach of most software engineers in industry, including even many formal methods researchers.
- Another challenge is that the use of formal methods usually costs more in time and human effort for analysis and design. One of the important reasons is the constant change of requirements during a software development process. When the initial high level specification is written, it is usually incomplete in terms of recording the user requirements. When it is refined into a lower level specification, the two specifications may not

satisfy the refinement rules, not necessarily because the lower level specification has errors, but rather because the high level specification is often not sufficiently complete. In this case, the high level specification needs to be modified or extended in order to reflect the user requirements, discovered during its refinement. Such a modification often occurs, not only to one level specification, but also to almost every level specification. This imposes a strong challenge to developers, both in psychology and in cost, especially when the project is under pressure from the market.

Having given the challenges to formal methods above, we should not deny the positive role of formal methods. In fact, formal methods have two advantages over informal ones. One is the high potential for automation in processing formal specifications due to their formally defined syntax and semantics. Another is that formal methods can work effectively for compact specifications. If one has experience reading research papers in software engineering or other areas, one will easily understand that reading a paper full of mathematical definitions and formulas, with less informal explanations, is much harder than reading a paper with a proper combination of informal explanations and small-scale formal descriptions (leaving necessary large-scale formal definitions in the appendix). Using formal notation in specifications has a similar effect on their readability. This is an important point about formal methods that has made us realize the importance of integrating formal methods with commonly used and comprehensible informal or semi-formal notations in software engineering. Formal notation can be used for the most critical and lower level components of a complex system, while a comprehensible notation can be adopted to integrate those formal definitions to form the entire specification, without losing the focus on *what to do*.

Furthermore, although formal verification may be difficult to be deployed directly in practice, its principles may be incorporated into existing practical techniques, such as testing, static analysis, and animation to achieve more effective verification and validation techniques. It is important to strike a good balance between rigor and practicality in integrated verification and validation techniques.

1.4 Formal Engineering Methods

Formal Engineering Methods, FEM for short, are the methods that support the application of formal methods to the development of large-scale computer systems. They are a further development of formal methods toward industrial application. I proposed to use this terminology for the first time in 1997 when organizing the first International Conference on Formal Engineering Methods (ICFEM) in Hiroshima [63] and continued to use it in many publications since then [76, 70, 74, 75, 64, 78, 69].

Formal engineering methods are equivalent neither to application of formal methods, nor to formal methods themselves. They are intended to serve as
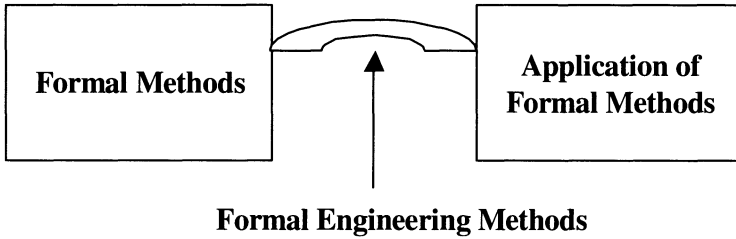
Fig. 1.5. An illustration of formal engineering methods

a *bridge* between formal methods and their applications, providing methods and related techniques to incorporate formal methods into the entire software engineering process, as illustrated in Figure 1.5. Without such a bridge, application of formal methods is difficult. The quality of the bridge may affect the smoothness of the formal methods technology transfer. Some types of bridges may make the transfer easier than others, so the important point is how to build the bridge.

Similar to formal methods, formal engineering methods are also aimed at attacking the problems in specification and verification of computer systems, but take more practical approaches. In principle, formal engineering methods should allow the following:

- Adopting specification languages that properly integrate graphical notation, formal notation, and natural language. The graphical notation is suitable for describing the overall structure of a specification comprehensibly, while the formal notation can be used to provide precise abstract definition of the components involved in the graphical representation. The interpretation of the formal definitions in a natural language helps understand the formal definitions. Many graphical notations have already been used for requirements analysis and design in practice, such as Data Flow Diagrams (DFDs) [23, 117], Structure Charts [15], Jackson Structure Diagrams [50, 16], and UML (Unified Modeling Language) [30, 18], but most of them are informal or semi-formal. This is the reality, but not necessarily a definitive feature of graphical notation. In fact, a graphical notation can also be treated as formal notation, as long as it is given a precise syntax and semantics. Compared with textual mathematical notation, a graphical notation is usually easier to read, but it usually takes more space than textual notation, and perhaps drawing diagrams is less efficient than typing in textual notation. Therefore, an appropriate integration can create a comfortable ground for utilizing the advantages of both graphical notation and formal notation.
- Employing rigorous but practical techniques for verifying and validating specifications and programs. Such techniques are usually achieved by integrating formal proof and commonly used verification techniques, such

as testing [108, 72, 91], reviews [94, 97], and model checking [49, 17]. The integrated techniques must take a proper approach to make good use of the strong points of the techniques involved and to avoid their weaknesses.

- Advocating the combination of prototyping and formal methods. A computer system has both dynamic and static features. The dynamic feature is shown only during the system operation, such as the layout of the graphical user interface, usability of the interface, and performance. The requirements for these aspects of the system are quite difficult to capture without actually running the system or its prototype. For this reason, prototyping – the development of an executable model of the system can be effective in capturing the user requirements for some of the dynamic features in the early phases of system development. The result of prototyping can serve as the basis for developing an entire system using formal methods, focusing on the functional behaviors of the system. Of course, sometimes prototyping can go along, in parallel, with the development using formal methods.
- Supporting evolution rather than strict refinement in developing specifications and programs [57, 109, 82, 73, 7]. Evolution of a specification, at any level, means a change, and such a change does not necessarily satisfy the strict refinement rules (of course, it sometimes does). The interesting point is how to control, support, and verify changes of specification during software development in a practical manner. Although some of these issues are still open to be resolved, they have been increasingly paid attention to by researchers.
- Deploying techniques for constructing, understanding, and modifying specifications. For example, effective techniques for specification construction can be achieved by integrating existing requirements engineering techniques with formal specification techniques [77], and techniques in simulation and computer vision can be combined to form visualized simulation to help specification understanding, and so on.

In summary, formal engineering methods embrace integrated specification, integrated verification, and all kinds of supporting techniques for specification construction, transformation, and system verification and validation. They can be simply described as

$$\begin{aligned} \textbf{FEM} = \ &\textbf{Integrated specification} + \\ &\textbf{Integrated verification} \ \ + \\ &\textbf{Supporting techniques} \end{aligned}$$

Note that formal engineering methods are a collection of specific methods, so we should not expect a single formal engineering method to cover all the features given previously.

1.5 What Is SOFL

SOFL, standing for *S*tructured *O*bject-Oriented *F*ormal *L*anguage, is a formal engineering method. It provides a formal but comprehensible language for both requirements and design specifications, and a practical method for developing software systems. The language is called *SOFL specification language*, while the method is called *SOFL method*. Unless there is the need of clear distinction, SOFL is used to mean either the language or the method or both throughout this book, depending on the context.

SOFL is designed by integrating different notations and techniques on the basis that they are all needed to work together effectively in a coherent manner for specification constructions and verifications. The SOFL specification language has the following features:

- It integrates Data Flow Diagrams [23], Petri nets [12], and VDM-SL (Vienna Development Method - Specification Language) [54, 55, 110]. The graphical notation Data Flow Diagrams are adopted to describe comprehensibly the architecture of specifications; Petri nets are primarily used to provide an operational semantics for the data flow diagrams; and VDM-SL is employed, with slight modification and extension, to precisely define the components occurring in the diagrams. A formalized Data Flow Diagram, resulting from the integration, is called *Condition Data Flow Diagram*, or CDFD for short. It is always associated with a *module* in which its components, such as processes (describing an operation), data flows (describing data in motion), and data stores (describing data at rest), are formally defined. In semantics, the CDFD associated with a module describes the behavior of the module, while the module is an encapsulation of data and processes, with an overall behavior represented by its CDFD. Furthermore, the use of a natural language, such as English, is facilitated to provide comments on the formal definitions in order to improve the readability of formal specifications [76, 41, 26].
- Condition data flow diagrams and their associated modules are organized in a hierarchy to help reduce complexity and to achieve modularity of specifications. Such a hierarchy is formed by decomposition of processes. A process is decomposed into a lower level CDFD and its associated module when the details of how to transform its input to output needs to be spelled out.
- *Classes* are used to model complicated *data flows and stores*. A store is like a file or database in many computer systems; it offers data that can be accessed by processes in a CDFD or by different CDFD in the hierarchy. The value of a store can be used and changed by processes. If the changes are made by processes at different levels, it will be difficult to control the changes. For this reason, a store can be modeled as an instance of a class. A class is a specification for its instances or objects that contains definitions of attributes and methods (similar to processes, but with constraints). Any change of the attributes of an instance must be made by its own methods.
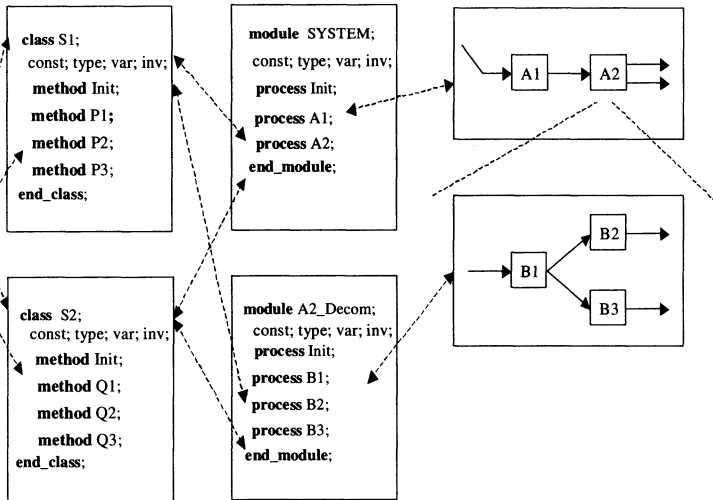
Fig. 1.6. An outline of a specification in SOFL

Modules and classes are similar in their internal structures, but different in the way used in specifications. A module represents a decomposition of a high level process and has an overall behavior. No instance can be derived from a module; therefore, a module cannot be used as a type to declare variables. On the other hand, objects may be instantiated from a class that may offer many individual behaviors, as services, and are used to model a data flow or store in CDFDs.

Figure 1.6 shows an outline of a specification in SOFL. The hierarchy of CDFDs and modules contains two CDFDs and associated modules. Each small rectangle in the CDFDs denotes a process, and each directed line represents a data flow. The CDFD involving processes A1 and A2 is the top level CDFD, corresponding to the module SYSTEM. In this module, the functions of A1 and A2 are formally defined. In addition, process Init is provided for the initialization of the local data stores (which are not given in this abstract figure) and necessary declarations are given. For some reason process, A2 is decomposed into the CDFD containing processes B1, B2, and B3, and its associated module, named A2_Decom, provides formal definitions of its processes, data flows, and so on. For the specification of processes in the hierarchy of CDFDs, classes S1 and S2 are defined; they may be used in both modules, SYSTEM and A1_Decom.

The SOFL method has the following features:

- It integrates *structured methods* and *object-oriented methods* for specification construction, in order to utilize their advantages and to avoid their disadvantages. Structured methods are a top-down approach by which the construction of a specification starts from the top level module, and then

proceeds by decomposing high level operations defined in the modules into low level modules. The structured methods are usually intuitive for requirements analysis and design, because their way of documentation is consistent with the way in which people think in developing and organizing large-scale projects, such as building a bridge, launching a rocket, or making an aircraft. On the other hand, object-oriented methods are basically a bottom-up approach to software development. In this approach the low level classes are first built, and then they are composed to form more complicated classes. Furthermore, an object-oriented approach is effective in achieving system properties, such as encapsulation of data and operations, inheritance, and polymorphism. These properties are very important in achieving the qualities of information hiding, software reuse, and maintainability. However, this approach may be less intuitive than structured methods for requirements analysis and design. The integration of these two different but related approaches in SOFL offers a way to effectively support functional decomposition and object composition. The specifications are easy to be translated into commercially object-oriented programming languages, such as C++ [61] and Java [22].

- It supports a *three-step approach* to developing formal specifications. Such a development is an *evolutionary* process, starting from an informal specification, to a semi-formal one, to finally a formal specification. The informal specification, usually written in a natural language, serves as the basis for deriving the semi-formal specification in which SOFL syntax, to a certain extent, is enforced. The formal specification is then derived from the semi-formal specification by formalization of the informal parts in the semi-formal specification.

By considering the roles of requirements and design specifications, SOFL advocates the idea that requirements specifications are written in a semi-formal manner, while design specifications need to be completely formal. The obvious reason for this is that requirements specifications are often used for communication between the user and the developer, which requires the comprehensibility of documentation, while the primary role of design specification is to provide an unambiguous ground for implementation. Furthermore, the construction of design specification requires study of requirements given in the requirements specifications, and formalization can greatly help in this regard.

An evolution of specification is a change, which can be a *refinement, extension,* or *modification* [66]. The evolution approach is suited to developing design specifications on the basis of semi-formal requirements specifications, since it usually results in many changes in the specifications. But for implementation from a design specification, refinement must be enforced, since we must make sure that the implementation does exactly same thing required by the design. For the details of this approach, see Chapter 14.

- It adopts *rigorous review* and *testing* for specification verification and validation. Specification verification aims to detect faults in specifications. Rigorous review is a technique resulting from the integration of formal proof and fault tree analysis, a method for safety analysis. The reviews must be done on a precise ground, and supported by a rigorous mechanism [67][68]. They are usually less formal than formal proof, but easy to conduct.
  Testing can be applied to both specifications and programs. Since some formal specifications are not executable, the testing needs a special technique [72]. The test cases used for specification can be reused for black-box testing of programs [91]. For the detailed discussions of these techniques, see Chapters 17 and 18.

When building a specific software system, the techniques supported by SOFL can be used with flexibility, depending on the application domain. For critical systems, such as safety- and security-critical systems, a profound use of formal notation, rigorous testing, and rigorous review are recommended. But for less critical systems, semi-formal notation and reasonably rigorous verification may be sufficient.

1.6 A Little History of SOFL

The initial development of SOFL was made at the University of Manchester in the United Kingdom in 1989, when I was studying for a doctoral degree in formal methods. The motivation was to integrate the most well-known formal method, VDM at that time, with traditional DFDs to support the application of formal methods in industry. I strongly believed, and still do now, that software development is not a pure mathematical process, although the relation between specifications and programs can be interpreted mathematically. It is, in fact, a highly disciplined *human* activity featured by creativity and constant changes, although it is likely supported by software tools. If any powerful method wants to be accepted by practitioners at large, it must provide a user-friendly interface and effective mechanism to facilitate the structuring of large-scale systems. On the other hand, informal methods that have been using in practice offer no guarantee for the quality of software systems. It was my belief that it is necessary to develop a kind of formal method from the *engineer's* point of view, and a *proper* combination of formal, semi-formal, and informal notations can possibly provide a good solution.

I chose VDM and DFDs for three reasons. One is that both are appropriate notations to describe "*what to do*" rather than "*how to do it*," but on different level. In DFDs this feature is reflected by focusing on data flows among processes (rather than control flows in algorithms), while in VDM it is featured by using pre- and postconditions for operation specifications. Another reason is that VDM lacks an effective and comprehensible structuring

mechanism to allow a large specification to be formed by integrating different operations. Although the notion *module* is used to organize operations in specifications, its expressive power and scale-up ability are limited. In addition to this weakness, the readability of large-scale specifications may not be satisfactory. However, it became quite clear to me after a period of study that VDM and DFDs are complementary in providing rigorous and comprehensible specifications, and that the notation for operation specification in VDM is well suited to describing specifications for processes used in DFDs. This provided the third reason for the integration. The language resulting from this research was called FGSL, standing for *F*ormal *G*raphical *S*tructured *L*anguage.

FGSL was evolved continuously later on, by combining my experiences gained from several projects on formal methods and safety-critical systems at the University of York, RHBNC of London University, Hiroshima City University, The Queen's University of Belfast, Oxford University, and Hosei University. It was an important step when FGSL was developed into SOFL by integrating the structured method and object-oriented method on the project titled "Formal Methods and Intelligent Software Engineering Environments" sponsored by the Ministry of Education, Culture, Sports, Science and Technology of Japan in 1996. It was an international joint project involving the researchers from several universities in Japan, USA, UK, and Australia. Since then, SOFL has been improved after being applied to the modeling or development of some critical systems and information systems on national and international projects [74, 75, 78, 62, 71].

1.7 Comparison with Related Work

It is quite difficult within a section to give a comprehensive comparison of SOFL with all the existing work on integration of formal methods and informal or semi-formal methods. To help the reader understand the commonality and difference between SOFL and other related work, we try to focus on the work that attempts to integrate model-oriented formal methods (e.g., VDM, Z, Alloy [48]) and semi-formal methods (e.g., data flow diagrams, UML).

From late 1980s more and more researchers began to realize the importance of combining formal and informal methods, and proposed several different approaches to integrating formal specification languages with informal notations (and associated methods). The approach taken by most researchers for integration is to use the Yourdon or the DeMarco approach to constructing data flow diagrams and their associated data dictionaries for expressing high level user requirements, and then to refine the data flow diagrams into formal specifications by defining data flows, necessary processes, and their integration with formal notation. The examples of this approach include Semmens and her colleagues' work on integrating Yourdon's data flow diagrams and Z [103], Bryant's work on Yourdon's method and Z [14], Plat and his colleagues' integration of data flow diagrams and VDM [96], and Fraser's work on data

flow diagrams and VDM [80]. In contrast to this approach, SOFL is aimed at achieving both the improvement of structuring mechanism in the VDM specification language for modularity and the comprehensibility of the ultimate specifications. This target is realized by incorporating classical data flow diagram notation into a formal specification language to provide a decompositional method for structuring system specifications and a graphical view for the system specifications. In this way, data flow diagrams are treated as part of formal specifications. Although adopting a rather different data flow model for describing computer systems, Broy and Stolen's FOCUS formalism [13] shares the idea of employing visual formal notation in specifications. However, the major difference between FOCUS and SOFL is that the former tries to provide a mathematical and logical foundation for the specification and refinement of interactive systems, while the latter emphasizes the techniques for incorporating formal specification and verification into the entire software development process to improve the quality of the software process and to achieve the practicality of formal methods.

Apart from the integration of formal methods and the structured method based on the data flow paradigm, much work has also been done in combining formal notations with the object-oriented paradigm or notation for concurrency to improve the rigor of object-oriented development or concurrent development. Examples of this approach include VDM++ [27], Object-Oriented Z [85], TCOZ [81, 25], and OCL [112], the Object Constraint Language of UML (Unified Modelling Language) [93, 18, 30]. Although SOFL also adopts object-oriented features, such as class and object, class inheritance, and polymorphism, it emphasizes a quite different development paradigm than UML in that the structured method is mainly used for user requirements analysis and abstract design specification in order to effectively capture the desired functions and the overall architecture of the system, while the object-oriented method is mainly used for detailed design and implementation to achieve good maintainability and reusability of the system. Another distinct feature of SOFL is that it emphasizes a balance between and compatibility with graphical notation and formal notation: it advocates the use of both formal and graphical notations for good readability and efficiency in constructing specifications, but does not encourage concentration on the use of only one of them.

Developing practical techniques for verification and validation of software systems based on formal specification and proof has also been an intensively researched area. The proposed techniques include *specification animation* [40, 89], *model checking* [17, 5], *specification-based testing* [108, 104, 113, 91, 92], and *software review, inspection, and analysis* [94, 87, 79, 21]. Since we take the view in SOFL that harmony among methods, tools, and human developers is the key to the success of software projects, we adopt the most practical techniques, software review and testing, for verification and validation, although the specific methods for review and testing may be different from traditional approaches. In our methods, we emphasize utilizing formal specification and

proof principle to achieve *rigor* for the practical review and testing techniques, as well as their supportability using software tools.

1.8 Exercises

1. Answer the following questions:
   a) What is the software life cycle?
   b) What is the problem with informal approaches to software development?
   c) What are formal methods?
   d) What are the major features of formal engineering methods?
   e) What is SOFL?
2. Explain the role of specification in software development.
3. Give an example of using a principle similar to formal methods to build other kinds of systems rather than software systems.

2

Propositional Logic

SOFL specifications usually involve both diagrams and formal textual definitions. The underlying languages for writing formal definitions are the classical propositional and predicate logics. These logics are also used for defining properties of specifications and for expressing conditions for specification verification. In this chapter, we introduce the propositional logic, and in the next chapter we explain predicate logic. Since SOFL adopts slightly different syntax of some logical operators for the sake of readability, the presentations in this and the next chapter will directly use SOFL syntax to be consistent with specifications discussed throughout the book.

Propositional logic deals with propositions, including representation, combination, and evaluation of and reasoning about propositions.

2.1 Propositions

A *proposition* is a statement that must be either true or false, but not both. A proposition can be represented by either a natural language sentence or a mathematical expression.

For example, the following statements are propositions:

- A tiger is an animal.
- An apple is a fruit.
- $3 + 5 > 10$.

The first and second propositions are true, but the third one is false under the usual interpretation of arithmetic symbols.

In contrast with these statements, the following statements are not propositions, since their truth values are not decidable.

- Are you happy?
- Let's go swimming.

Table 2.1. Propositional operators

operator	read as	priority
not	not	highest
and	and	
or	or	
=>	implies	
<=>	is equivalent to	lowest

We use **bool** to represent the boolean type, a set of the truth values: **true** and **false**. That is,

   bool = {**true**, **false**}.

To facilitate manipulation of propositions, they are usually denoted by symbols. For instance, the propositions given previously are denoted by the symbols:

- P: A tiger is an animal.
- Q: An apple is a fruit.
- R: $3 + 5 > 10$.

2.2 Operators

Simple propositions can be combined using *propositional operators*, which are sometimes also called *logical operators*, to form *compound propositions*. The propositional operators used in SOFL are given in Table 2.1.

   The table gives the operator's symbol, how it is read, and its precedence when applied to form compound propositions. Using these operators, we can combine the previously introduced propositions P, Q, and R to form the compound proposition:

   P => Q **and** R <=> **not** P **or** Q **and** R .

The evaluation of this proposition may start with those of Q **and** R and **not** P. The results of these two constituent propositions can then be used for the evaluation of P => Q **and** R and **not** P **or** Q **and** R. Finally, the entire proposition is evaluated based on the intermediate results. To explicitly emphasize the priority of the propositional operators, parentheses can be used. Thus, this proposition is equivalent to

   (P => (Q **and** R)) <=> ((**not** P) **or** (Q **and** R)) .   (1)

For convenience in our discussions, we use the term *propositional expression* or *expression* to mean a single proposition or compound proposition. A single proposition is also called *atomic proposition*, because it is the smallest unit to form propositional expressions. For example, in the expression (1) given above, P, Q, and R can be atomic propositions, and their compositions formed by using the propositional operators are compound propositions.

2.3 Conjunction

A conjunction is a propositional expression whose principal operator is **and.** For example,

x > 5 **and** x < 10

shows a conjunction, stating that x is bigger than 5 and smaller than 10. The general form of a conjunction is:

P **and** Q ,

where P and Q are constituent propositions. The complete interpretation of this conjunction is given by the *truth table*:

P	Q	P and Q
true	true	true
true	false	false
false	true	false
false	false	false

The first two columns, from the left, give all possible truth values P and Q can take, and the third column gives the results of the conjunction P **and** Q. The conjunction P **and** Q is **true** only when both P and Q are **true**, and **false** when one of them is **false**.

For example, we can easily derive the following from this truth table:

true and true <=> true
false and true <=> false
false and false <=> false

Note that the operator <=> has the same function as the equality symbol =, but is used only between logical expressions, meaning that both sides have the same truth value. When appropriate, the symbol = is also used alternatively to express the equality between logical expressions in SOFL.

2.4 Disjunction

A disjunction is a propositional expression whose principal operator is **or.** It is usually intended to represent a condition that holds as long as one of its constituent propositions holds. For example, suppose x is an integer; the proposition

  x > 5 or x < 3

presents a condition that x is either bigger than 5 or smaller than 3. Let P and Q be two propositions, the disjunction of P and Q is written as

  P or Q .

The meaning of operator **or** is defined by the truth table:

P	Q	P or Q
true	true	true
true	false	true
false	true	true
false	false	false

A disjunction is **true** when one of its constituent propositions is **true**, and **false** when both its constituent propositions are **false**. As an example, we derive the following from this truth table:

  true or true <=> true
  true or false <=> true
  false or false <=> false

2.5 Negation

A negation is a propositional expression whose principal operator is **not**. Let P be a proposition, the negation built of P is

  not P .

The negation **not** P is true if and only if P is false, as defined by the true table:

P	not P
true	false
false	true

The negation of a proposition represents an opposite state of the proposition. For example, if P denotes the condition x > 5, then **not** P will denotes the condition: x <= 5, where <= means "less than or equal to."

2.6 Implication

An implication is a propositional expression that uses the operator $=>$ to connect its constituent propositions. Let P and Q be propositions, then the implication

$$P => Q$$

expresses a statement that P is stronger than Q. P is called *antecedent* and Q is called *consequent.* The complete definition of the implication is given by the truth table:

P	Q	P => Q
true	true	true
true	false	false
false	true	true
false	false	true

Basically the implication $P => Q$ means that if P is true, it must ensure that Q is true; if P is false, Q can be either true or false. In other words, $P => Q$ is true means that the values taken by P and Q are "reasonable." A daily life related example may help understand this point. Let P denote the proposition: "John works hard", and Q the proposition "John receives an award." Suppose we apply common sense that people working hard can receive an award. Then, the truth of the implication $P => Q$ points to two situations:

(1) John works hard and John receives an award.
(2) John does not work hard and it does not matter whether John receives an award or not.

In case (1), it is reasonable that John receives an award as he works hard. This does not seem to be difficult to understand based on common sense. In case (2), as John does not work hard, he may or may not receive an award. We do not say, in this case, that he will definitely receive no award because he does not satisfy the precondition: working hard. Rather, we are not interested in what will happen since the precondition is not true.

2.7 Equivalence

An equivalence is a propositional expression indicating that its constituent propositions are of the same strength. Let P and Q be propositions. Then, the equivalence

$$P <=> Q$$

means that P and Q are equivalent in the sense that their truth values are the same. The truth table for the equivalence is

P	Q	P <=> Q
true	true	true
true	false	false
false	true	false
false	false	true

Equivalence represents the equality between truth values. As mentioned before, the symbol $=$ may also be used in SOFL alternatively to express the equality between truth values, or more generally between logical expressions.

2.8 Tautology, Contradiction, and Contingency

A *tautology* is a special proposition that evaluates to true in every combination of the truth values of its constituent propositions. Consider the proposition

    P **or not** P

as an example. No matter what truth values P takes, this proposition always evaluates to true.

On the other hand, if a proposition evaluates to false in every combination of its constituent propositions, it is known as *contradiction*. For example, the proposition

    P **and not** P

is a contradiction. Apparently, a contradiction is a negation of a tautology. Tautology and contradiction are important concepts that will be used in formal *proof* to be introduced later in this chapter.

A proposition that is neither a tautology nor a contradiction is known as *contingency*. Regarding tautologies and contradictions as extreme cases, contingencies are the most common propositions to be used. For example, the proposition

    P $=>$ Q **and** R

is a contingency, because the result of its evaluation depends on the truth values of P, Q, and R.

2.9 Normal Forms

There are two important normal forms: *disjunctive normal form* and *conjunctive normal form.* A disjunctive normal form is a special kind of disjunction in which each constituent propositional expression, usually known as *disjunctive clause,* must be a conjunction of atomic propositions or their negations.

Let `P_1`, `P_2`, ..., `P_n` be conjunctions of atomic propositions or their negations, respectively. Then, the expression

   `P_1 or P_2 or ... or P_n`

is a disjunctive normal form. The characteristic of such a disjunctive normal form is that it evaluates to true as long as one of the disjunctive clauses evaluates to true.

A *conjunctive normal form* is a special kind of conjunction in which each constituent propositional expression, usually called *conjunctive clause,* is a disjunction of atomic propositions or their negations. Suppose each of `Q_1`, `Q_2`, ..., `Q_m` is a disjunction of atomic propositions or their negations, then

   `Q_1 and Q_2 and ⋯ and Q_m`

is a conjunctive normal form. Such a conjunctive normal form is featured by the property that it evaluates to false as long as one of its conjunctive clauses evaluates to false.

2.10 Sequent

A sequent is an assertion that a *conclusion* can be deduced from *hypotheses.* A hypothesis is given as a propositional expression, presenting an assumed property or fact. A conclusion is also a propositional expression, which is expected to be supported by its hypotheses. Let `P_1`, `P_2`, ..., `P_n` be hypotheses and `Q` a conclusion. Then a sequent representing that `Q` is deduced from `P_1`, `P_2`, ..., `P_n` is written as

   `P_1, P_2, ..., P_n ⊢ Q`

where ⊢ is called a *turnstile.* For example, the sequent

   `P and Q ⊢ P`

states that the conclusion `P` can be deduced from the hypothesis `P and Q`.

Note that the validity of a sequent is not necessarily guaranteed by only conforming to its syntax. Usually, it needs to be proved in some way. That is, whether the conclusion of a sequent can be deduced from its hypotheses needs a formal proof. Without such a proof, we have no evidence to support the validity of the sequent.

Table 2.2. The truth table for a proof

P	Q	not P and Q	P => Q
true	true	false	-
true	false	false	-
false	true	true	true
false	false	false	-

2.11 Proof

A proof is a process (or activity or evidence) to show that the conclusion can be established from its hypotheses in a sequent. Two ways can be adopted to prove the validity of a sequent. One is by establishing a truth table for the sequent. Once the hypotheses of the sequent have been evaluated, the conclusion needs be evaluated only in those rows where the hypotheses are all true. Consider the sequent

    not P **and** Q $\vdash$ P => Q

as an example. A truth table for the proof of its validity is given in Table 2.2. In this truth table four combinations of truth values of P and Q are given, but there is no need to evaluate the conclusion P => Q for the first two and the last rows, since the evaluations of the hypothesis **not** P **and** Q for those rows are false.

Using truth table for proof is rather straightforward. However, the difficulty will be increased greatly when the number of constituent propositions becomes bigger. If a sequent to be proved involves 10 constituent propositions, then the number of all their combinations will be $2^{10} = 1024$. Obviously using truth table for proof in this case is not convenient and efficient at all.

Another way of doing proof, which is known as *natural deduction*, can help reduce this problem. A deduction is a way to derive a conclusion from the hypotheses by applying appropriate *inference rules* available in the logic.

2.11.1 Inference Rules

An inference rule is composed of two parts: *a list of premises* and a *conclusion*. A premise is a propositional expression, and so is a conclusion. A rule is usually written in the form:

$$\frac{premise_1, \ premise_2, \ ..., \ premise_n}{conclusion} \boxed{name}$$

where the *name* is for reference in a proof; it usually indicates what this rule is about.

This rule states that the truth of the conclusion is the consequence of the truth of the premises. In other words, when the premises are true, then

so is the conclusion. In a formal proof, a rule can be applied only when its premises are true. Note that the list of premises can be empty, meaning that the conclusion is a tautology. In this case, the rule is known as *axiom*, and can be applied whenever necessary.

To make the introduction of the inference rules clear, we divide them into groups, each being associated with a specific kind of propositional expression (e.g., conjunction, disjunction, etc). we first describe the basic rules for conjunction, disjunction, negation, implication, and equivalence, respectively, and then introduce the properties of propositional expressions, which can be proved by applying these basic rules. These properties can also serve as derived rules for inference.

2.11.2 Rules for Conjunction

Three basic rules for a conjunction are available. These rules ether describe how to introduce the **and** operator from its constituent propositions or how to eliminate the **and** operator from a conjunction. They are named as **and**-*intro*, **and**-*elim1*, and **and**-*elim2*, respectively.

$$\frac{P,\ Q}{P\ \textbf{and}\ Q}\ [\textbf{and-}intro] \qquad\qquad \frac{P\ \textbf{and}\ Q}{P}\ [\textbf{and-}elim1]$$

$$\frac{P\ \textbf{and}\ Q}{Q}\ [\textbf{and-}elim2]$$

Suppose P and Q are true; then, we can apply the [**and**-*intro*] rule to prove the truth of P **and** Q. On the other hand, if P **and** Q is already known to be true, the rule [**and**-*elim1*] can be applied to deduce P, and the rule [**and**-*elim2*] applied to deduce Q. It is easy to prove the validity of these rules by means of truth tables.

2.11.3 Rules for Disjunction

Similarly to the rules for conjunction, there are also three basic rules for disjunction: [**or**-*intro1*], [**or**-*intro2*], and [**or**-*elim*].

$$\frac{P}{P\ \textbf{or}\ Q}\ [\textbf{or-}intro1] \qquad\qquad \frac{Q}{P\ \textbf{or}\ Q}\ [\textbf{or-}intro2]$$

$$\frac{P\ \textbf{or}\ Q,\ P \vdash R, Q \vdash R}{R}\ [\textbf{or-}elim]$$

The [**or**-*intro1*] and [**or**-*intro2*] rules indicate how to introduce the **or** operator from its constituent propositions, while the [**or**-*elim*] rule shows how the **or** operator is eliminated.

The validity of the [**or**-*intro1*] and [**or**-*intro2*] rules can be easily proved by means of truth tables, but the [**or**-*elim*] rule is a little more complicated. The reason is that in the list of premises, sequents are involved. To ensure the validity of the sequents $P \vdash R$ and $Q \vdash R$, subsidiary proofs are necessary. These proofs are done by deducing R, the conclusion of the rule, from either P or Q.

2.11.4 Rules for Negation

The basic rules for negation are two, named [**not**-*intro*] and [**not**-*elim*].

$$\frac{P}{\text{not not } P} \text{ [\textbf{not}-\textit{intro}]} \qquad\qquad \frac{\text{not not } P}{P} \text{ [\textbf{not}-\textit{elim}]}$$

The [**not**-*intro*] rules states that if P is true, then its double negation is also true. The [**not**-*elim*] rule shows the case opposite to the [**not**-*intro*] rule.

2.11.5 Rules for Implication

There are two basic rules for implication as follows:

$$\frac{Q}{P => Q} \text{ [=>-\textit{intro}]} \qquad\qquad \frac{P => Q, P}{Q} \text{ [=>-\textit{elim}]}$$

According to the truth table defining implication, whenever Q is true, no matter what truth value P takes, the implication $P => Q$ evaluates to true. This definition leads to the establishment of the rule [=>-*intro*] . On the other hand, the rule [=>-*elim*] shows how to eliminate the operator =>. The reason for this rule to be valid is that when the antecedent P is true, the only value for Q to take is true, in order to ensure the truth of the implication $P => Q$.

2.11.6 Rules for Equivalence

For an equivalence $P <=> Q$, there are three basic inference rules describing how to introduce and eliminate the equivalence operator <=>.

$$\frac{P => Q, Q => P}{P <=> Q} \text{ [<=>-\textit{intro}]} \qquad \frac{P <=> Q}{P => Q} \text{ [<=>-\textit{elim1}]}$$

$$\frac{P <=> Q}{Q => P} \text{ [<=>-elim2]}$$

These rules are built based on the fact that $P <=> Q$ is equivalent to the two implications: $P => Q$ and $Q => P$.

2.11.7 Properties of Propositional Expressions

There are many properties about propositional expressions, but we give only the important ones that are used in this book. Each property is given as a rule.

Proposition 1. *A conjunction, disjunction, or equivalence is commutative. That is,*

$$(1) \quad \frac{\text{P and Q}}{\text{Q and P}} \ [\textbf{and-}comm]$$

$$(2) \quad \frac{\text{P or Q}}{\text{Q or P}} \ [\textbf{or-}comm]$$

$$(3) \quad \frac{\text{P} <=> \text{Q}}{\text{Q} <=> \text{P}} \ [<=>\text{-}comm]$$

The rules given in this **proposition** are bidirectional: the expressions above and below the double line imply each other. For example, the [**and-**comm] rule is equivalent to the following two rules:

$$\frac{\text{P and Q}}{\text{Q and P}} \ [\textbf{and-}comm1] \qquad \text{and} \qquad \frac{\text{Q and P}}{\text{P and Q}} \ [\textbf{and-}comm2]$$

The other two rules can be interpreted similarly.

These properties can be formally proved by applying the basic rules introduced previously. Let us first consider the proof of the commutativity of conjunction as an example of how a proof is actually constructed. In fact, proving this property is equivalent to proving the validity of the following sequents:

(1) P **and** Q ⊢ Q **and** P
(2) Q **and** P ⊢ P **and** Q

As the proof of (2) is similar to that of (1), we only give the proof for (1). To provide a comprehensible presentation of the proof, we use a box to enclose the proof. Such an approach is known as *boxed proof*. The boxed proof for the sequent (1) is given in Table 2.3.

The keyword **from** starts a hypothesis, and **infer** is followed by a conclusion deduced by applying appropriate rules whose names are given on the right of the box. The intermediate results of the proof are marked by numbers, indicating the number of the steps that have been taken in the proof.

Table 2.3. A proof for commutativity of conjunction

```
from P and Q
1    P                          [and-elim1]
2    Q                          [and-elim2]
infer Q and P                   [and-into]
```

Table 2.4. A proof for the commutativity of disjunction

```
from  P or Q
1  from  P
     infer  P or Q              [or-intro1]
     infer  Q or P              [or-comm]
2  from  Q
     infer  P or Q              [or-intro2]
     infer  Q or P              [or-comm]
infer   Q or P                  [or-elim](h, 1, 2)
```

The proof of the commutativity of disjunction is a little more complicated, because we need to consider two cases: when each of the constituent propositions is true. As in the case of conjunction, proving this property is equivalent to proving the following two sequents:

(1)   P or Q ⊢ Q or P
(2)   Q or P ⊢ P or Q

Again, we only give the proof of (1), and leave the proof of (2) to the reader for exercise. The boxed proof for (1) is given in Table 2.4.

The proof starts with the hypothesis P or Q. Since the truth of this disjunction depends on the truth of one of its constituent propositions, we need to consider the two cases independently. One is when P is true and the other is when Q is ture. In both cases we can deduce Q or P, by applying appropriate rules. Therefore, the disjunction Q or P is established by applying the rule [or-elim], based on the hypothesis, step 1, and step 2, which is indicated by the reference [or-elim](h, 1, 2)].

The proof of the commutativity of equivalence can be done similarly. The reader can construct the proof as an exercise.

Proposition 2. *A conjunction, disjunction, implication, or equivalence is associative. That is,*

$$\frac{\text{P and (Q and R)}}{\text{(P and Q) and R}} \text{ [and-}ass]\qquad\frac{\text{P or (Q or R)}}{\text{(P or Q) or R}} \text{ [or-}ass]$$

$$\frac{\text{P => (Q => R)}}{\text{(P => Q) => R}} \text{ [=>-}ass]\qquad\frac{\text{P <=> (Q <=> R)}}{\text{(P <=> Q) <=> R}} \text{ [<=>-}ass]$$

These properties are given in four rules, and their proofs can be done similarly to those of the properties given in Proposition 1. For brevity, we will omit all the proofs for the properties to be introduced in this section.

Proposition 3. *Conjunctions and disjunctions are distributive over each other.*

$$\frac{\text{P and (Q or R)}}{\text{(P and Q) or (P and R)}}\ [\textbf{and-or-}dist]$$

$$\frac{\text{P or (Q and R)}}{\text{(P or Q) and (P or R)}}\ [\textbf{or-and-}dist]$$

The rule [**and-or-**dist] describes the equivalence between the expressions P and (Q or R) and (P and Q) or (P and R), while the rule [**or-and-**dist] states the equivalence between P or (Q and R) and (P or Q) and (P or R).

Proposition 4. *An implication is equivalent to a disjunction, i.e.,*

$$\frac{\text{P} => \text{Q}}{\text{not P or Q}}\ [=>\textbf{-or-}equiv]$$

This rule provides a way of transformation between an implication and an equivalent disjunction.

Proposition 5. *Negations, conjunctions, and disjunctions satisfy the de Morgan's laws:*

$$\frac{\text{not (P and Q)}}{\text{not P or not Q}}\ [\textbf{and-}deM] \qquad \frac{\text{not (P or Q)}}{\text{not P and not Q}}\ [\textbf{or-}deM]$$

By the rule [**and-**deM], the negation of a conjunction can be transformed into an equivalent disjunction of the negations of the constituent propositional expressions, and vice visa. The rule [**or-**deM], on the other hand, allows the negation of a disjunction to be transformed into an equivalent conjunction of the negations of the constituent expressions. You will find that de Morgan's laws are necessary in the transformation of propositional expressions into *disjunctive normal forms*, which are to be introduced next.

Proposition 6. *A propositional expression is equivalent to a disjunctive normal form or a conjunctive normal form.*

In other words, a propositional expression can be transformed into an equivalent disjunctive normal form or a conjunctive normal form. Such a transformation is done by repeatedly applying distributivity rules [**and-or**-*dist*] and [**or-and**-*dist*], as well as the de Morgan's laws introduced previously.

For example, the propositional expression

P_1 and not (**P_2 and P_3**)   (1)

can be transformed into a disjunctive normal form by taking the following steps:

P_1 and not (**P_2 and P_3**) <=>
P_1 and (**not P_2 or not P_3**) <=>
(**P_1 and not P_2**) **or** (**P_1 and not P_3**) <=>
P_1 and not P_2 or P_1 and not P_3

The derived disjunctive normal form is composed of the two conjunctions:

(1) **P_1 and not P_2**
(2) **P_1 and not P_3**

We can take similar approach to transforming a propositional expression into an equivalent conjunctive normal form. Since the notion of conjunctive normal form is not used in this book, we do not explain it further with examples. The reader can conduct this transformation as an exercise.

2.12 Exercises

1. Explain the notions:
   a) proposition
   b) conjunction
   c) disjunction
   d) negation
   e) implication
   f) equivalence
   g) tautology
   h) contradiction
   i) contingency
   j) sequent
   k) rule
   l) proof
2. Give a truth-table proof for each of the properties:
   a) P, Q ⊢ P **and** Q
   b) P **and** Q ⊢ Q

    c) $P \vdash P$ **or** Q

    d) P **or** Q, $P \Rightarrow R$, $Q \Rightarrow R \vdash R$

    e) $P \vdash$ **not not** P

    f) $Q \vdash P \Rightarrow Q$

    g) $P \Rightarrow Q$, $Q \Rightarrow P \vdash P \Leftrightarrow Q$

3. Give a boxed proof for each of the properties:

    a) P **and** $(Q$ **and** $R) \vdash (P$ **and** $Q)$ **and** R

    b) P, Q, $Q \Rightarrow R \vdash P$ **and** R

    c) **not** $(P$ **or** $Q) \vdash$ **not** Q

    d) P **or** $Q \vdash$ **not** (**not** P **and not** $Q)$

4. Transform each of the following propositional expressions into a disjunc-
tive normal form:

    a) P **and not** (**not** Q **and** $R)$

    b) P **and** $(Q \Rightarrow R) \Leftrightarrow W$

    c) $(P$ **or** $Q)$ **and** $(R$ **or** $W)$

    d) **not** $(P \Rightarrow Q)$ **or** (**not** P **and** $Q)$

    e) $P \Leftrightarrow Q$ **and** $Q \Leftrightarrow R$

3

Predicate Logic

Predicate logic is an extension of the propositional logic to deal with the statements that may apply to many objects. The propositional logic introduced in the preceding chapter allows us to make statements about specific objects, but it does not allow us to make statements applicable to a collection of objects. For example, in the propositional logic we can make the statement: John studies hard, but we cannot make statements like x studies hard, where x is one of the students John, Michael, Steven, or Paul, since the truth value of the statement cannot be determined until x is bound to a specific name. Note that when x is bound to a specific name, the entire statement can be either true or false. For example, the statement John studies hard can be true whereas the statement Steven studies hard can be false.

Furthermore, sometimes we may want to describe a property that every member in a collection of objects must satisfy, such as "every player in the club is excellent". These are known as *universal* statements. Some other times, we may wish to state that at least one member satisfies a particular property, without necessarily knowing which member it is (or they are). For example, "there is an excellent player in the club". Such a statement is known as *existential* statement.

In this chapter we introduce the predicate logic by explaining what predicates are, how predicates are combined to form compound predicates, how proofs can be done on predicates, and how predicates can be used to express specifications or properties. This logic is also extended to provide a reasonable treatment of *undefined* predicates, which may be employed in specifications.

3.1 Predicates

A *predicate* is a *truth-valued* function. A function is a mathematical abstraction of an important concept: mapping between two sets of values. The set of values to which the function can apply is known as the *domain* of the func-

Table 3.1. The basic types available in SOFL

name	symbol	values
natural numbers including zero	**nat0**	0, 1, 2, 3, ...
natural numbers	**nat**	1, 2, 3, 4, ...
integers	**int**	..., -2, -1, 0, 1, 2, ...
real numbers	**real**	..., -1.25, ..., 0, ..., 1.25, ...
boolean	**bool**	**true**, **false**

tion, while the set containing the images of the mapping from the domain is known as the *range* of the function.

In SOFL, *types* are often used as the domain and range of functions. A type usually means a set of values, possibly with a set of operations. The name, symbol, and values of the basic types available in SOFL are given in Table 3.1.

A function is defined by giving its *signature* and *body*. The signature is usually composed of three parts: *function name*, *domain*, and *range*. The function name is a unique identity, distinguishing it from other functions. The domain is written in parentheses following the function name, and the range is separated from the right parenthesis by a colon. The body of the function is given after the double equality symbol ==. For example, the function square over integers is defined as follows:

square (x: **int**): **int**
== x * x

In this definition, square is the function name; the **int** used to declare the parameter x denotes the domain; and the **int** on the right is the range. The body of the function is given as a product of the parameter x. Note that the name for the parameter of the function can be chosen freely, without changing the definition of the function. For instance, we can choose y instead of x so that the square function can be defined as

square (y: **int**): **int**
== y * y .

A function may have more than one parameter, so the definition of a function in general looks like

f(x_1: T_1, x_2: T_2, ..., x_n: T_n): T
== E(x_1, x_2, ..., x_n) ,

where T_i (i=1..n) are types; the domain of the function is the product type: T_1 * T_2 * ··· * T_n; and E(x_1, x_2, ..., x_n) is an expression containing variables x_1, x_2, ..., x_n.

For example, a function known as multi is defined as

multi(x1: **nat**, x2: **nat**, x3: **nat**): **nat**
== x1 * x2 * x3

multi(x1, x2, x3) yields the product of the three parameters.

If we restrict the range of a function to the boolean type **bool**, then the function is known as a predicate. For example, the function is_big defined below is a predicate.

is_big(x: **int**): **bool**
== x > 2000

As we mentioned in the beginning of this chapter, a predicate is different from a proposition in that the predicate cannot evaluate to truth values unless its parameters are bound to specific values in the domain. Therefore, we cannot say that is_big(x) is true or false, since x is not bound to a specific value in the domain **int**. However, if we let x take the values 2000, 2004, 2008, 1996, respectively, we will be able to obtain the following four propositions:

is_big(2000)
is_big(2004)
is_big(2008)
is_big(1996)

According to the definition of the predicate is_big, it is apparent that is_big(2000) is false; is_big(2004) is true; is_big(2008) is true; and is_big(1996) is false. Similarly to functions, predicates may also allow multiple parameters, so the definition of a predicate in general has the form

P(x_1: T_1, x_2: T_2, ..., x_n: T_n): **bool**
== E(x_1, x_2, ..., x_n) .

Compound predicates may be formed by using the logical operators and other defined predicates that may contain function applications. Consider the predicate compare as an example.

compare(x_1: **int**, x_2: **int**): **bool**
== is_big(x_1) **and** square(x_1) >= square(x_2)

The predicate is defined in terms of the predicate is_big and the predciate square(x_1) >= square(x_2), in which the function square is applied. It states that when the parameters x_1 and x_2 are bound to values in the domain of the predicate, and they satisfy the condition is_big(x_1) **and**

square(x_1) >= square(x_2), the proposition compare(x_1, x_2) will evaluate as true; otherwise, it will evaluate as false.

Predicates can be combined by using the propositional operators: **and**, **or**, **not**, =>, and <=>, in the exactly same way as for propositions to form compound predicates. Thus,

> is_big(x_1) **and** compare(x_1, x_2) => x_1 > x_2
> is_big(x_1) **or** compare(x_1, x_2)
> **not** is_big(x_1)
> is_big(x_1) => compare(x_1, x_2)
> is_big(x_1) <=> is_big(x_2)

are all compound predicates.

3.2 Quantifiers

In the predicate logic we use two quantifiers. One is *the universal quantifier* and the other is *the existential quantifier*.

3.2.1 The Universal Quantifier

A predicate allows us to make a statement that is applicable to a set of objects, such as is_big defined in the previous section. However, if we wish to make a statement that requires a set of objects to satisfy a property, using only the notion of predicate introduced so far may not be sufficient to form concise predicate expressions. For example, if we want to make the statement that 2004, 2008, 2012, and 2016 all satisfy the condition given by the predicate is_big, we can write it as

> is_big(2004) **and** is_big(2008) **and** is_big(2012) **and** is_big(2016) .

However, this expression is long and cumbersome. To make such an expression concise, we introduce a notion known as *universal quantifier*, represented by the keyword **forall**. For example, the above predicate expression is written as follows using the universal quantifier:

> **forall**[x : {2004, 2008, 2012, 2016}] | is_big(x)

This statement states that for any element x in the set {2004, 2008, 2012, 2016}, the proposition is_big(x) evaluates to true. Such a predicate is called a *universally quantified* predicate or expression. In SOFL specifications, the general form of a universally quantified expression is written as

> **forall**[x: X] | P(x) .   (1)

In this expression, **forall** is the universal quantifier; x is known as the *bound variable*; x: X is called a *constraint* and X is a *bound set*; and P(x) is known as the *body* of the quantified expression, which can be a single predicate, a compound predicate, or another quantified predicate expression.

The quantified expression states that, for any value x in set X, P(x) is satisfied. It is true if all the elements in the set X satisfy P(x), and false otherwise. Note that P(x) may contain other variables, in addition to the bound variable x. If those variables are not bound variables in any quantified expression in P(x), we call them *free variables*. For example, in the expression

forall[x: int] | square(x) > y ,

y is a free variable. This quantified expression cannot evaluate to a truth value due to the existence of the free variable y, unless y is bound to a specific value in its type.

In the general form of the universally quantified expression (1), if the body P(x) does not contain free variables, the expression will become a proposition, since, in that case, its truth value can be decided. For example, the expression

forall[x: {2004, 2008, 2012, 2016}] | is_big(x)

is actually a proposition. It is true because every element of the set {2004, 2008, 2012, 2016} satisfies the predicate is_big. If, however, we change this expression to

forall[x: {1996, 2000, 2012, 2016}] | is_big(x) ,

then this expression is false, because both 1996 and 2000 do not satisfy is_big.

3.2.2 The Existential Quantifier

When making a statement that requires at least one element of a set to satisfy a property, we usually need to use a disjunction. For instance,

is_big(1996) **or** is_big(2000) **or** is_big(2004) **or** is_big(2008)

states that at least one element of the set {1996, 2000, 2004, 2008} satisfies is_big. By using the existential quantifier, represented by the keyword **exists**, this expression is written as

exists[x: {1996, 2000, 2004, 2008}] | is_big(x) .

Such an expression is known as an *existentially quantified* expression. The general form of an existentially quantified expression is written as:

$$\textbf{exists}[\textsf{x: X}] \mid \textsf{P(x)} \, , \hspace{6cm} (2)$$

except that the name of the quantifier in expression (2) is different from that of the quantifier in the universally quantified expression (1) (all of the other parts share the same names). The existentially quantified expression is true if at least one element of set $\textsf{X}$ satisfies the predicate $\textsf{P(x)}$, and false otherwise. As in a universally quantified expression, if the body $\textsf{P(x)}$ involves free variables, the existentially quantified expression becomes a predicate, not a proposition.

When constructing formal specifications using predicate logic, as we will explain later in this chapter, it is sometimes useful to be able to express "there exists exactly one," rather than "there exists one." This is represented by the extended existential quantifier **exists!**. For example, the predicate

$$\textbf{exists!}[\textsf{x: int}] \mid \textsf{x} = 0$$

states that there exists exactly one integer zero.

3.2.3 Quantified Expressions with Multiple Bound Variables

A quantified expression also allows multiple bound variables. In general, a universally quantified expression with $\textsf{n}$ bound variables is written as

$$\textbf{forall}[\textsf{x\_1: X\_1, x\_2: X\_2, ..., x\_n: X\_n}] \mid \textsf{P(x\_1, x\_2, ..., x\_n)} \, .$$

If $\textsf{X\_1, X\_2, ..., X\_n}$ are all the same set, then this expression can also be written as

$$\textbf{forall}[\textsf{x\_1, x\_2, ..., x\_n: X\_1}] \mid \textsf{P(x\_1, x\_2, ..., x\_n)}$$

Likewise, an existentially quantified expression has the form

$$\textbf{exists}[\textsf{x\_1: X\_1, x\_2: X\_2, ..., x\_n: X\_n}] \mid \textsf{P(x\_1, x\_2, ..., x\_n)}$$

or

$$\textbf{exists}[\textsf{x\_1, x\_2, ..., x\_n: X\_1}] \mid \textsf{P(x\_1, x\_2, ..., x\_n)} \, ,$$

if $\textsf{X\_1, X\_2, ..., X\_n}$ are all the same. For example, the following expressions are quantified expressions with multiple bound variables:

$$\textbf{forall}[\textsf{x\_1, x\_2, x\_3: nat}] \mid \textsf{multi(x\_1, x\_2, x\_3)} >= \textsf{x\_1}$$
$$\textbf{exists}[\textsf{x\_1: int, x\_2: nat}] \mid \textsf{square(x\_1)} + \textsf{x\_2} > 10$$

3.2.4 Multiple Quantifiers

Multiple quantifiers can be used to express more complicated predicate expressions. To make expressions with multiple quantifiers concise, the quantifiers can be combined with bound sets. For example, the expressions

> **forall**[x: **nat**] | **forall**[y: **nat**] | P(x, y)
> **forall**[y: **nat**] | **forall**[x: **nat**] | P(x, y)
> **forall**[x: **nat**, y: **nat**] | P(x, y)
> **forall**[x, y: **nat**] | P(x, y)

all mean the same thing. Such a combination is also applicable to the existentially quantified expressions. However, when both the universal quantifier and existential quantifier are used in an expression, the combination becomes more complicated. Let us consider the expression

> **forall**[x: **nat**] | **exists**[y: **nat**] | y > x

as an example. It states that for any natural number, there must exist another greater natural number. This is obviously true. Note that we cannot simply exchange the universal quantifier and the existential quantifier in this expression, since the changed expression

> **exists**[y: **nat**] | **forall**[x: **nat**] | y > x

means different thing: there exists a natural number that is greater than every natural number, which is apparently false.

To reduce the need for parentheses and avoid unnecessary confusion, we adopt the convention that the body of a quantified expression is considered to extend as far to the right as possible. So the expression

> **forall**[x: **nat**] | (x > z **and** (**exists**[y: **nat**] | y > x))

can be written as

> **forall**[x: **nat**] | x > z **and exists**[y: **nat**] | y > x .

3.2.5 de Morgan's Laws

Just as de Morgan's laws for propositions, there are also de Morgan's laws for quantified expressions. These laws are especially useful for proofs, to be introduced later in this chapter.

Proposition 7. *The quantified expressions satisfy de Morgan's laws. That is,*

$$\frac{\textbf{forall}[x: X] \mid P(x)}{\textbf{not } (\textbf{exists}[x: X] \mid \textbf{not } P(x))} \; [\textbf{exists-}deM]$$

$$\frac{\textbf{not } (\textbf{forall}[x: X] \mid P(x))}{\textbf{exists}[x: X] \mid \textbf{not } P(x)} \; [\textbf{forall-}deM]$$

These laws are quite straightforward. The law **exists-**deM implies that the statement "P(x) holds for every element in X" is equivalent to saying that "P(x) does not hold for some element in X is false." The law **forall-**deM states that the statement "P(x) does not hold for every element in X" is equivalent to the statement "P(x) does not hold for some element in X."

3.3 Substitution

Substitution is an operation that changes a predicate by substituting a variable or expression for a free variable in the predicate. Such a substitution allows us to change the subject of the predicate. Let P be a predicate; we use P[x/y] to denote the predicate obtained by substituting variable x for every *free* occurrence of y in P. The following are examples of substitution:

(x > 5 **and** y > x)[t/x] <=> (t > 5 **and** y > t)
(10 > 20)[y/x] <=> (10 > 20)
(x < 20 + y)[(2+z)/y] <=> (x < 20 + (2 + z))

In the first case, variable x is substituted by t, and the structure of the predicate and other variables remain unchanged. In the second substitution, the predicate (an extreme case of predicate) is not changed at all, since x is not involved in the predicate. The third case shows a substitution of a variable by an arithmetic expression.

When making a substitution to a predicate involving quantified expressions, we must not substitute any bound variables in the predicate. Consider the following substitutions:

(1) (**forall**[x: **nat**] | x + 1 > 0)[y/x] <=>
     (**forall**[x: **nat**] | x + 1 > 0)
(2) (**exists**[y: **nat**] | y > x **and** y < x + 15)[5/x] <=>
     (**exists**[y: **nat**] | y > 5 **and** y < 5 + 15)

The first case shows that an attempt to substitute a variable for a bound variable in a predicate has no effect. In other words, such a substitution does not change the predicate at all. In the second substitution each occurrence of

the free variable x is substituted by 5. Note that a substitution must not cause confusion between free and bound variables. For instance, the substitution

(y > 10 **and exists**[x: **nat**] | x > y)[x/y]

results in the following predicate with confusion between the free and bound variables:

x > 10 **and exists**[x: **nat**] | x > x

A contradiction x > x is introduced as the result of this substitution, which is obviously not what we want. The reason for such a problem is that the substituting variable shares the same identifier with the bound variable; both are x. It is the general principle that the substituting variable should be different from any bound variables occurring in the predicate, especially when a potential confusion may occur.

This conflict problem can be resolved by first changing the identifiers of the relevant bound variables in the predicate and then performing the substitution. For example, before carrying out the substitution

(y > 10 **and exists**[x: **nat**] | x > y)[x/y] ,

we first change the bound variable x to i, and then do the substitution, which results in the predicate

x > 10 **and exists**[i: **nat**] | i > x .

Substitutions can be done *sequentially* more than once. We use P[x/y][t/x] to denote the predicate resulting from first substituting x for occurrences of y in P and then substituting t for occurrences of x in the predicate P[x/y]. The following is an example of a sequential substitutions:

(y > 10 **and exists**[i: **nat**] | i > y)[x/y][t/x] <=>
(x > 10 **and exists**[i: **nat**] | i > x)[t/x]   <=>
(t > 10 **and exists**[i: **nat**] | i > t)

A single substitution can be extended to allow multiple substitutions. We use P[x/y, t/z] to denote the predicate resulting from *simultaneously* substituting x for occurrences of y and substituting t for occurrences of z. The following example shows a multiple substitution:

(y > z + 10 **and exists**[i: **nat**] | i > y + z)[x/y, t/z] <=>
(x > t + 10 **and exists**[i: **nat**] | i > x + t)

Similarly, we can extend this notation to P[x1/y1, x2/y2, ..., xn/yn].

3.4 Proof in Predicate Logic

Since the predicate logic is an extension of the propositional logic, all the inference rules available in the propositional logic are applicable to predicates. However, there is no rule in the propositional logic to deal with quantifiers, so the existing rules are in general not sufficient to handle proofs in the predicate logic. It is necessary to introduce rules for reasoning about quantifiers in the predicate logic.

3.4.1 Introduction and Elimination of Existential Quantifiers

The existentially quantified expression **exists[x: X] | P(x)** states that P(x) holds for some element x in X. Therefore, if we know that a value, say m, is a member of X and P(x)[m/x] holds, we will definitely be able to assert that **exists[x: X] | P(x)** is true. Based on this observation, the rule for introducing an existential quantifier is formed as follows:

$$\frac{\textsf{m inset X, P(x)[m/x]}}{\textsf{exists[x: X] | P(x)}} \: [\textbf{exists-}\textit{intro}] \: ,$$

where m **inset** X means that x is a member of X; the membership operator **inset** is discussed in detail in Section 8.3.2 of Chapter 8.

On the other hand, if **exists[x: X] | P(x)** is known to be true, and an *arbitrary* value, say m, belongs to X and the expression Q can be induced from P(x)[m/x], then we can claim that Q holds. This idea is reflected by the following rule for the elimination of existential quantifiers:

$$\frac{\textsf{exists[x: X] | P(x), m inset X and P(x)[m/x]} \vdash \textsf{Q}}{\textsf{Q}} \: [\textbf{exists-}\textit{elim}] \: ,$$

m is arbitrary

Note that m must not occur as a free variable in the predicate Q and must be different from any variables occurring in the earlier proof steps.

3.4.2 Introduction and Elimination of Universal Quantifiers

If we know any *arbitrary* value, say y, in X such that P(y) holds, then we can definitely conclude that **forall[x: X] | P(x)** holds, based on the meaning of the universally quantified expression. This idea is reflected by the rule

$$\frac{\textsf{y inset X} \vdash \textsf{P(y)}}{\textsf{forall[x: X] | P(x)}} \: [\textbf{forall-}\textit{intro}] \: ,$$

As with the existential quantifier, a rule of eliminating a universal quantifier is also available:

Table 3.2. A boxed proof

from	y inset X, not P[y/x]	
1	**exists[x: X] | not P(x)**	**exists-***intro*(h)
infer	**not (forall[x: X] | P(x))**	**forall-***deM*(1)

$$\frac{\textbf{forall[x: X] | P(x), m inset X}}{\textsf{P(m/x)}} \; [\textbf{forall-}\textit{elim}]$$

This rule defines that if **forall[x: X] | P(x)** holds and m is a member of X, then the truth of P(x)[m/x] can be safely claimed.

Now let us look at an example of proof that applies the rules introduced above. Of course, we may also need other existing inference rules, including de Morgan's laws, given in Section 3.2.5. The proposition to be proved is given as the sequent

    y inset X, not P[y/x] ⊢ not (forall[x: X] | P(x)) .

The boxed proof of this sequent is given in Table 3.2.

3.5 Validity and Satisfaction

Validity and satisfaction are two important properties of the predicate logic.

Definition 1. *A predicate is valid if it evaluates to true for whatever values of the free variables involved.*

For example, let x be a variable over the type **int**. Then the predicate

    x > 0 or x <= 0

is valid, because it evaluates to **true** no matter what integer the variable x takes. Sometimes we are interested in whether a predicate is true for some values. If so, we say the predicate is *satisfiable*.

Definition 2. *A predicate is satisfiable if it evaluates to **true** for some values of the free variables involved. Otherwise, the predicate is unsatisfiable.*

For instance, let x be a variable over the type **int**, the predicate

    x > 10

is satisfiable, since it evaluates to **true** for some integers, say 15. On the other hand, the predicate

$x > 10$ **and** $x < 10$

is unsatisfiable, since it evaluates to **false** for whatever integers bound to x.

The notions of valid, satisfiable, and unsatisfiable predicates correspond in fact to those of tautology, contingency, and contradiction in the propositional logic, respectively. A valid predicate is similar to a tautology; a satisfiable predicate is similar to a contingency; and an unsatisfiable predicate is similar to a contradiction.

3.6 Treatment of Partial Predicates

It is possible that a predicate may not yield a truth value for some values bound to its free variables (arguments). Such a predicate is called *partial predicate*. For example, the predicate $x \; / \; y > 20$ involves a division between two real numbers x and y. If y is not zero, the truth value of this predicate can be determined. However, if y is equal to zero, the result of division $x \; / \; y$ is *undefined*, which leads to the entire predicate being undefined.

Furthermore, as the reader will see in Chapters 17 and 18, the predicate expressions that need to be evaluated for reviews or testing may involve undefined variables. In such cases, how to evaluate the expressions will become a problem: the result should be true, false, or undefined. To deal with this problem, we need a logical system in which undefinedness is treated as a "special value" for operations. It is treated as a "value" because it can join operations as an operand, and it is "special" because it is in fact not a real value, but represents a situation of no value or a situation when a variable is bound to a value of the wrong type. Fortunately, VDM has provided a proper logical system to handle partial predicates by extending the two-valued truth tables of propositional operators into three-valued truth tables. SOFL also adopts this extension for interpreting formal specifications.

In the extended truth tables, the absence of value, i.e., undefinedness, is represented by the keyword **nil**. Since nine cases must be considered now for each operator, the truth tables are presented in a compact square style. The extended truth table for conjunction is:

(and)	true	nil	false
true	**true**	**nil**	**false**
nil	**nil**	**nil**	**false**
false	**false**	**false**	**false**

In order to be distinguished from operands, the operator **and** is in parentheses. The extension is made in a way that a result is given whenever possible. For example, the conjunction

nil and false

yields **false**, as it is always the result of the evaluation of this conjunction, no matter what truth value the **nil** can possibly be. That is, if the position of **nil** is **true**, the result of the evaluation is **false**, and so is the result of the evaluation when the position of **nil** is **false**. If the evaluation of a conjunction cannot yield a truth value due to the lack of truth values, reflected by the involvement of **nil**, the result of the evaluation must be **nil**. For example,

> **nil and true <=> nil**
> **nil and nil   <=> nil**

The same principle is also applied to extend the truth tables for disjunction, negation, implication, and equivalence, which are given below.

(or)	true	nil	false
true	true	true	true
nil	true	nil	nil
false	true	nil	false

(not)	
true	false
nil	nil
false	true

(=>)	true	nil	false
true	true	nil	false
nil	true	nil	nil
false	true	true	true

(<=>)	true	nil	false
true	true	nil	false
nil	nil	nil	nil
false	false	nil	true

As pointed out by Jones in his book titled *Systematic Software Development Using VDM* [55], the extended logical system does not inherit some properties of the classical logic. For example,

> P **or not** P

is a tautology in the classical logic, but it is not true in extended logic because it yields no truth value when P is undefined. Therefore, in order to apply all the rules of the classical logic in proof, one must make sure that every predicate expression involved is defined. However, if only evaluations of expressions are required for necessary arguments, they can be performed by applying the extended truth tables, with no need for using the inference rules. Since proof is rarely employed in SOFL for verification of systems, this problem is not a major concern.

3.7 Formal Specification with Predicates

This section explains how the predicate logic can be used to define functional requirements. As the detail of this technique will be introduced in Chapter 4, the description in this section is kept simple. The reader is expected to get, through examples, a preliminary idea about the use of the logic for functional specifications of software systems.

A predicate is usually used to describe a condition, but its role may be interpreted in a number of ways in a specification, depending on the way it is used. Let us consider the predicate

$x > 5$ **and** $x < 10$

as an example. It can be used as a *guard* condition for determining the subsequent action or definition. It can also be used to express a functional requirement that the expected operation generates values satisfying this condition. This is similar to the following statement:

my student finishes his homework.

This proposition can serve as a condition to decide whether my student will be given a full mark or not, but it may also be used to express a requirement to my student for completing his homework. The reader will see more detailed discussions on this issue in Chapter 4. For the moment, as long as one understands the point that predicates can be used for writing both guard conditions and functional requirements, he or she is good to proceed to the next chapter.

3.8 Exercises

1. Answer the following questions:
    a) What is the similarity and difference between a predicate and a function?
    b) What is the difference between a universally quantified expression and existentially quantified expression?
    c) What is a substitution?
    d) What is a valid predicate?
    e) What is a satisfiable predicate?
    f) What is a partial predicate?
2. Which of the following quantified predicate expressions are propositions?
    a) **forall[x: int]** $| x > 5$ **and** $x < 10$
    b) **exists[x: int]** $| y > x$ **and** $y < x + 10$
    c) **forall[x, y: real]** $| x + y > x - y$

    d) **forall[x, y: real]exists[z: real]** | x + y > z

    e) **exists[x: int]forall[y: int]** | x * y > z

3. Evaluate the substitutions:

    a) (x > y + z => y < x)[t/x]

    b) (**forall[x, y: nat0]** | x < z **and** z < y => x < y)[m/y, t/z]

    c) (**exists[x, y, z: nat]** | x * y > z => x > z **and** y > z **and** b > c)[a/x, b/y, c/b]

4. Give proofs for the properties (assuming all the involved predicates are defined):

    a) **forall[x: X]** | P(x) ⊢ **not exists[x: X]** | **not** P(x)

    b) x **inset** X, **forall[y: X]** | y > 15 ⊢ **exists[z: X]** | z > 15, where X is a subset of **nat0**.

5. Which predicates are true according to the extended truth tables?

    a) x > y **and** y / 0 > 5 <=> **false**

    b) x > y **and** y > x <=> **nil**

    c) **true or nil** <=> **nil**

    d) **false or nil** <=> **false**

    e) **false** => **nil** <=> **nil**

    f) **true** => **false** <=> **nil**

    g) **true** => **nil** <=> **false**

    h) **true** <=> **false** <=> **nil**

    i) **false** <=> **nil** <=> **true**

4

The Module

This chapter introduces the most important component of SOFL specifica-
tions: the *module*. A specification is composed of a set of related modules in a
hierarchical fashion. Each module is a functional abstraction: it has a behavior
represented by a graphical notation, known as *condition data flow diagram*,
and a structure to encapsulate *data* and *processes* used in the condition data
flow diagram. Each data item is defined with an appropriate type and each
process is defined with a formal, textural notation based on the predicate logic
introduced in chapter 3.

4.1 Module for Abstraction

When building a software system, the essential concern is what function should
in the first place be provided by the system. An effective way to gain the under-
standing of the system function is *abstraction and decomposition*. Abstraction
is a principle for extracting the most important information from implemen-
tation details. The result of an abstraction is usually a concise specification
of the system, reflecting all the *primarily important* functions without *unnec-
essary details*. The understanding of this notion can be helped by studying
the simplified ATM (Automated Teller Machine) example. Suppose an ATM
is required to have the following functions:

(1) Provide the buttons *show the balance* and *withdraw* for selection.
(2) Insert a cash card and supply a password.
(3) If *show the balance* is selected, the current balance is given.
(4) If *withdraw* is selected, the amount of the money to withdraw is properly
      provided.
(5) The requested amount of money must be supplied in cash.

This list gives a functional abstraction of the desired system: it contains only
the functions of interest, described abstractly, and does not focus on the de-

tailed issues, such as what the buttons should look like and what the format of password is, and so on.

In an abstract specification the *dependency* between required functions may exist *implicitly*. For example, to perform function (3), function (1) must be performed so that the current balance can be provided; to ensure that function (4) is provided correctly, function (2) must be carried out beforehand so that only the permitted person can access the account. Sometimes, such a dependency may need to be defined *explicitly*. For example, as a requirement, function (2) may need to be performed before actions (3), (4), and (5). If this is not the case, the implementation of the system may end up performing action (4) before performing action (2), something which is obviously *unsafe*.

Abstraction may have different levels, and the high level abstraction may contain less information than the low level abstraction. For example, if we refine action (4) to take the possibility of a password mismatch into account, the functional specification can be written as follows:

(4') If *withdraw* is selected and the password is correct, the amount
    of the money to withdraw is provided; otherwise, if the password
    is wrong, a message for reentering the correct password must
    be given.

This may be considered as a concrete version of the abstract specification in (4), since it provides more detailed information about the required function. Of course, it may also be regarded as an abstraction of another lower level specification.

In order to avoid any potential misunderstanding, an abstract specification of a system must not be ambiguous. However, as we have described in Chapter 1, this problem will inevitably arise if informal languages are used to write the specification. For example, it is not clear what the phrase password is correct means in the above functional specification (4'). If we extend this specification to explain the meaning of phrases like this one, we will probably end up with a *long*, and probably more *complex*, documentation. To deal with this problem, formal notation can help greatly.

To allow functional abstractions at various levels and to help achieve the comprehensibility of formal specifications, SOFL employs condition data flow diagrams for the abstraction of data flow relations between processes performing specific behaviors, and employs predicate logic-based formal notation for the abstraction of data items and processes occurring in the diagrams. The concept of condition data flow diagram, including data flows and processes, and the formal notation for defining the components of the diagrams, are introduced from the next section in this chapter. Conceptually a module has the following structure:

ModuleName
condition data flow diagram
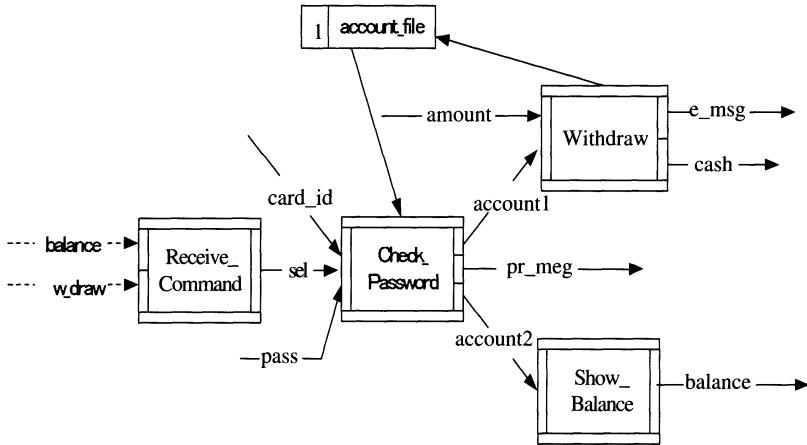
Fig. 4.1. The CDFD modeling a simplifed cash dispenser

Specification of the components

The components, and their syntax, of a module are gradually introduced in this chapter, starting from the next section.

4.2 Condition Data Flow Diagrams

A condition data flow diagram, CDFD for short, is a directed graph that specifies how processes work together to provide functional behaviors. Before proceeding to the detail of CDFD, let us try to get a preliminary idea by looking at the example of modeling the ATM with a CDFD, as shown in Figure 4.1.

Each box in this diagram denotes a process, such as Receive_Command and Check_Password, which describes an operation: it takes inputs and produces outputs. Each directed line with a labeled name denotes a data flow: the name indicates the nature of the data while the line gives direction of the data flow. The box with number 1 and the identifier account_file is known as a *data store* or *store*, which represents the data at rest, such as a file or database.

The diagram conveys the functional requirements given in section 4.1 and the dependency relations between the functions represented by the processes in the diagram. The selection of balance, denoting the command of *show the balance*, or w_draw, denoting the command of *withdraw*, is handled by the process Receive_Command. This process then generates a data flow sel to indicate which command has actually been selected, and passes this information to the process Check_Password. When the requested cash card card_id and password pass are provided, this process will check whether the provided account exists in the system account database account_file, and if so, whether
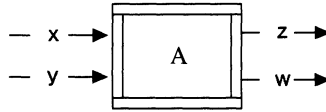
Fig. 4.2. A simple process

the password **pass** is the same as that of the account. If these pieces of information are confirmed, the process **Check_Password** will pass the account information, denoted by **account1** and **account2**, respectively, to either the process **Withdraw** or the process **Show_Balance**, depending upon the value of data flow **sel**. If security is, however, not confirmed, an error message **pr_meg** will be issued. The process **Withdraw** updates **account_file** by reducing the requested **amount** from the current balance of **account1** if the amount is less or equal to the current balance. However, if this is not the case, the process will generate an error message to show that the amount requested is invalid. The process **Show_Balance** takes the confirmed account denoted by **account2**, which is the same as **account1** in content, and displays the current balance of the account, which is represented by the data flow **balance**.

In contrast with the informal functional specification given in Section 4.1, the functional abstraction expressed by the CDFD is obviously more comprehensible in modeling the dependency relations among processes. To completely define the CDFD, however, all the processes, data flows, and stores must be defined precisely, in a proper manner. To achieve this goal, the predicate logic is adopted. From the next section, we describe all the possible components of CDFDs and the techniques for their formal definitions.

4.3 Processes

One of the most essential components is known as *process*. A process performs an action, task, or operation that takes input and produces output. To model the variety of operations, a process can take several different forms. Figure 4.2 shows a simple form of process.

The process **A** is composed of five parts: *name*, *input port*, *output port*, *precondition*, and *postcondition*. The name **A** of the process is given in the center of the box. The input port is denoted by the narrow rectangle on the left part of the box, which receives the input data flows **x** and **y**. The output port is given on the right part of the box, similar to the input port, to connect to the output data flows **z** and **w**. The upper part of the box, a narrow rectangle, denotes the precondition, while the lower part of the box represents the postcondition. The precondition of a process is a condition which the inputs are required to meet, while the postcondition is a condition which the outputs are required to satisfy. Before understanding the notions

of pre and postconditions in detail, it is essential to study how the process transforms its inputs to outputs operationally.

Briefly speaking this process transforms the *input data flows*, or *inputs* for short, x and y to the *output data flows*, or *outputs* for short, z and w. Note that the identifiers like x, y, z, and w are known as *data flow variables*, but when bound to specific values, they will represent the concrete data flows. In this case, we say that the data flows through variable x and y are *available*, or simply data flows x and y are available. We will discuss the notion of data availability in detail in Section 4.4. But for now this interpretation is sufficient to help us proceed to an explanation of the operational semantics of a process.

Precisely speaking, the operational semantics of this process is interpreted by the sequence of activities:

1. when *both* the input data flows x and y are available, the process is enabled, but it will not execute until the output data flows z and w become unavailable.
2. the execution of the process *consumes* the input data flows x and y, and *generates* the output data flows z and w.

Note that the availability of input data flows is not the only condition for executing a process. In fact, the execution requires both the availability of the input data flows and the unavailability of the output data flows. It is also important to understand the fact that only one of the input data flows x and y becomes available does not enable the process. Furthermore, after the execution of the process, all the input data flows are *consumed*; that is, they become unavailable.

We must emphasize that only understanding the operational meaning of a process is not sufficient for understanding the function of the process precisely. For example, how are the input data flows x and y used to generate the output data flows z and w? To answer questions like this, we need to provide a textual specification that describes the relation between the input data flows and output data flows. Such a specification is mainly composed of a precondition and a postcondition; both are predicate expressions. The precondition is a necessary condition that must be met by the input data flows in order for the process to be executed correctly. In other words, if the precondition is not satisfied by the input data flows, no correct output data flows are guaranteed. The postcondition shows a condition that the output data flows must satisfy after the execution of the process. Usually, in a postcondition the relation between the input data flows and output data flows are defined. Thus, how the input data flows are used to generate the output data flows can be seen clearly.

For example, the process A in Figure 4.2 is specified as:

process A(x: Ti_1, y: Ti_2) z: To_1, w: To_2
pre P(x, y)
post Q(x, y, z, w)
end_process

For consistency in documentation, **process** and **end_process** are a pair of keywords to mark the start and end of a process specification, respectively, and the process name A, and the input and output data flow variables must be kept the *same* as they appear in the graphical representation. Every data flow variable, including input and output data flow variables, must be declared with a type, such as **nat**, **int**, **real**. The declarations of the input data flow variables are given within parentheses following the process name A, and the declarations of the output data flow variables are given after the parentheses. Each variable declaration has the form

> *variable*: *type*

and different declarations are separated by a comma. For example, the input data flow variable x is declared with the type Ti_1 and y with Ti_2, and their declarations are separated by a comma. Likewise, the output data flow variables z and w are declared with the types To_1 and To_2, respectively.

The order of the declarations of data flow variables must be consistent with the order of their appearances in the corresponding graphical notation: *the top-down order of appearances in the graphical notation corresponds to the left-right order of declarations in the textual specification.* Thus, we can avoid potential confusions in understanding processes. For example, the specification of process A in the following form

> **process** A(y: Ti_2, x: Ti_1) z: To_1, w: To_2

is illegal because x appears in a higher position than y in the graphical notation, which is inconsistent with the left-right order in the textual specification.

pre is a keyword indicating the start of the precondition of the process, and the keyword **post** indicates the start of the postcondition. P(x, y) is the precondition of the process A, possibly involving x and y. Q(x, y, z, w) is the postcondition that possibly involves both the input variables x and y, and the output variables z and w, since it defines the relation between inputs and outputs. As an example, let us assume that all the variables x, y, z, and w are integers, and the process A does addition and subtraction based on x and y. Then the process A can be specified as

> **process** A(x: **int**, y: **int**) z: **int**, w: **int**
> **pre** x > 0 **and** y > 0
> **post** z = x + y **and** w = x - y
> **end_process**

The precondition states that both x and y must be greater than zero in order to assure a correct execution of this process. The postcondition requires that the

output data flow z be equal to the addition of x and y, and w the subtraction of y from x, after the execution of the process.

The declarations of input or output variables can be grouped together if they share the same type. For example, both x and y are the variables of type **int**, and so are z and w. Therefore, we can group x and y together, and z and w together in the specification of the process A to adopt the following form:

> **process** A(x, y: **int**) z, w: **int**
> **pre** x > 0 **and** y > 0
> **post** z = x + y **and** w = x - y
> **end_process**

This form is different from the previous one *only* in syntax, but not in semantics. A comma must be used to separate different variables in a group. Note that we must not group any input and output variables together, as they must be written in different places in the specification: input variables are given within the parentheses whereas the output variables are given after the parentheses.

Sometimes we may not want to impose any specific constraint on the input data flows of a process. That is, any input data flows of their types are legal inputs to this process. In this case, we let the precondition be the boolean value **true**, as in

> **process** A(x, y: **int**) z, w: **int**
> **pre true**
> **post** z = x + y **and** w = x - y
> **end_process**

Likewise, the same thing can be done for the postcondition of the process, as in

> **process** A(x, y: **int**) z, w: **int**
> **pre** x > 0 **and** y > 0
> **post true**
> **end_process**

According to this, any output data flows, as a result of the execution of the process, will satisfy the specification.

Sometimes we may not care what exactly a process does. For example, some university may want to give a special grant to a distinguished researcher for whatever research he or she wishes to conduct. In such a case, this university may require no precondition and postcondition for the researcher's work. A process of this kind is specified with both pre and postcondition being **true**, as in
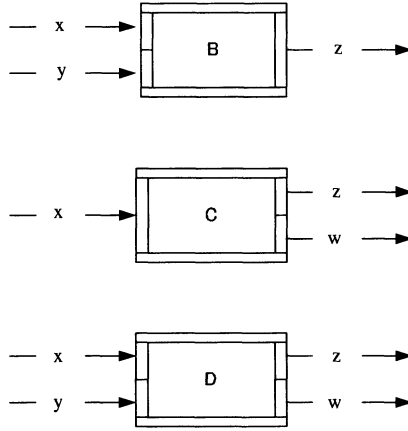
Fig. 4.3. Processes with multiple ports

> **process** A(x, y: **int**) z, w: **int**
> **pre true**
> **post true**
> **end_process**

For the sake of simplicity, SOFL allows the omission of the pre and postcondition parts in this special case. Thus, the above process A can be written as

> **process** A(x, y: **int**) z, w: **int**
> **end_process**

Of course, we can omit either the precondition or the postcondition if it is **true**, such as in

> **process** A(x, y: **int**) z, w: **int**
> **post** z = x + y **and** w = x - y
> **end_process**

or

> **process** A(x, y: **int**) z, w: **int**
> **pre** x > 0 **and** y > 0
> **end_process**

In other words, the absence of a precondition or a postcondition means that it is defined as **true**.

A process may have multiple input ports and/or output ports holding multiple groups of input and/or output data flows. Figure 4.3 illustrates various

forms of such a process. The process B has two input ports receiving the data flows x and y, and one output port holding the data flow z. When either x or y is available, process B takes x or y, but not both, as input, and produces z as output. The short horizontal line between the two input ports denotes an *exclusive relation* between the two groups of data flows in the sense that only one of them can be consumed in producing the output data flow. It is worth noting that this does not preclude situation in which both x and y are available. In such a situation, process B will choose non-deterministically one of x and y, say x, as its input for an execution. As the result of this execution, the data flow x is consumed, but the availability of y remains unchanged, and z is made available. Since y is still available, process B is enabled again, but its execution will not start until z is consumed by another process. Once z is made unavailable, process B will start an execution that consumes y and generates another z.

The formal specification of process B may be given as

> **process** B(x:Ti_1 | y: Ti_2) z: To_1
> **pre** P(x, y)
> **post** Q(x, y, z)
> **end_process**

The vertical line between the declarations of x and y in the textual specification corresponds to the horizontal line in the graphical representation of process B, denoting the exclusive relation between x and y. Since there is a possibility that either x or y, but not both, is available, and this availability will enable process B, we need to give the precondition as a *disjunction* of clauses, each involving either x or y but not both, in order to avoid an unexpected situation in which the precondition of the process can never be evaluated to true. For example, we can specifically define process B as follows:

> **process** B(x: **int** | y: **int**) z: **int**
> **pre** x > 0 **or** y > 0
> **post** z = x + 1 **or** z = y - 1
> **end_process**

Since the output variable z is defined based on one of the input variables x and y, the postcondition also needs to be a disjunction in this case. Otherwise, the postcondition may never be satisfied by any output data flows. Let us consider the following specification, which is derived from the modification of process B, as an example:

> **process** B(x: **int** | y: **int**) z: **int**
> **pre** x > 0 **and** y > 0
> **post** z = x + 1 **and** z = y - 1
> **end_process**

Suppose x, but not y, is available and greater than zero; then the precondition becomes

x > 0 **and nil** ,

which is equivalent to **nil** (as explained in Chapter 3). Similarly, the postcondition also evaluates to **nil** due to the unavailability of y.

However, when no specific constraints are needed for *all* the possible input or output data flows, we will just let the precondition or postcondition be the boolean value **true**, as in

> **process** B(x: **int** | y: **int**) z: **int**
> **pre** **true**
> **post** z = x + 1 **or** z = y - 1
> **end_process**

But, if there is a specific constraint on x but not on y in the precondition, for example, then we need to adopt a special symbol extended from the truth value **true**, to accurately reflect this condition, as illustrated by the process specification

> **process** B(x: **int** | y: **int**) z: **int**
> **pre** x > 0 **or true**(y)
> **post** z = x + 1 **or** z = y - 1
> **end_process**

Where **true**(y) is a predicate (not a truth value) defined as follows:

> **true**(y) = **true** if y is available
> **true**(y) = **nil** if y is unavailable

Thus, when x is available while y is not, the **true**(y) is **nil**, and the entire precondition can still be satisfied by x.

Now let us discuss the other forms of processes given in Figure 4.3. Process C takes x as input and produces either z or w, but not both. The short horizontal line between the two output ports related to z and w denotes an *exclusive relation* between z and w. Note that, which of z and w is generated can be *nondeterministic*; but it can also be *deterministic*, depending on how the process C is formally specified. For example, we can specify process C as

> **process** C(x: **int**) z: **int** | w: **int**
> **pre** x > 0
> **post** z = x + 1 **or** w = x * 2
> **end_process**

Upon the availability of input data flow x, this process generates either z that equals x + 1, or w that doubles x, but indicates no definite condition for choosing the generation of z or w. Therefore, it describes a nondeterministic situation. The exclusive relation between the availability of z and w is reflected using the vertical line between the declarations of z and w, as with the input data flows described previously.

If the generation of x or w needs to be deterministically defined, specific conditions must be given. For example, the specification

> **process** C(x: **int**) z: **int** | w: **int**
> **pre** x > 0
> **post** x < 10 **and** z = x + 1 **or** x >= 10 **and** w = x * 2
> **end_process**

defines z when x < 10, and otherwise w when x >= 10.

Since only one of z and w can be generated as a result of executing the process, the postcondition of process C is given as a disjunction of clauses, each involving only one of output variables z and w. Note that the postcondition cannot be a conjunction; otherwise, the semantics of the formal specification of the process will conflict with its operational semantics. For instance, suppose the operator **or** is changed to **and** in the postcondition of process C, forming

> **process** C(x: **int**) z: **int** | w: **int**
> **pre** x > 0
> **post** z = x + 1 **and** w = x * 2
> **end_process**

This postcondition will be impossible to satisfy with only one of the output data flows z and w. Note that only one of z and w is generated as the result of executing process C.

The process D in Figure 4.3 takes either x or y as input, and generates either z or w as output. But which input data flow is used to produce which output data flow is not precisely given by the graphical symbol of the process. This detailed level definition can be given in the formal specification of the process. For example, process D is specified as follows:

> **process** D(x: Ti_1 | y: Ti_2) z: To_1 | w: To_2
> **pre** P(x, y)
> **post** x <> **nil and** Q_1(z, x) **or** y <> **nil and** Q_2(y, w)
> **end_process**

When x is available, denoted by the expression x <> **nil**, output data flow z should be made available as a result of executing the process. When y is available, denoted by y <> **nil**, w should be generated. When both x and y are available, the generation of z or w is nondeterministic. In general, within
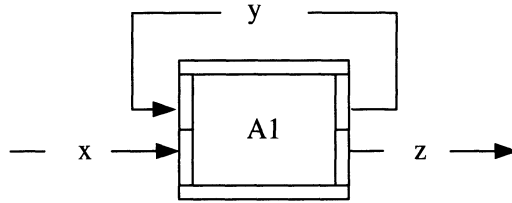
Fig. 4.4. A process with a data flow loop

a process specification there is no easy way to define exactly which of the available input data flows connecting to different ports needs to be consumed for executing the process. For this reason, the availability of more than one input data flows going to different input ports of a process should be avoided, unless it is absolutely necessary.

A process may have a data flow loop: a data flow is both the input and the output of the process. For instance, process A1 in Figure 4.4 has a data flow loop formed by data flow y. This process describes a counter. The initial value of the counter is provided by data flow x with the value 0. Then data flow y is generated as an increment of x, and y keeps increasing until its value reaches 100. In this case, data flow z is generated to take 100 as the final result of this loop. This process can be specified as follows:

> **process A1(y: nat0 | x: nat0) y: nat0 | z: nat0**
> **pre** x = 0 **or true**(y)
> **post** y = x + 1 **or**
>   ˜y < 100 **and** y = ˜y + 1 **or**
>   ˜y >= 100 **and** z = ˜y
> **end_process**

In the postcondition, the variable ˜y, decorated with the tiled symbol ˜, denotes the input data flow y, while y denotes the output data flow y. As we will see in Section 4.5 of this chapter, the same technique will also be used to represent data stores in the postconditions of processes.

A process may have no input or output data flow. The graphical representations of such a process is given in Figure 4.5. Process E has no input data flow; it has only an output data flow z. Such a process is intended to provide a data flow like z whenever requested by other processes in a CDFD, so its operational semantics is described as follows: *the process is always enabled to execute; once the output data flows are consumed, the new output data flows are generated.* In other words, the output data flows of such a process are always available for use. We call this kind of process *source process* or *source.* The formal specification of process E has the form:
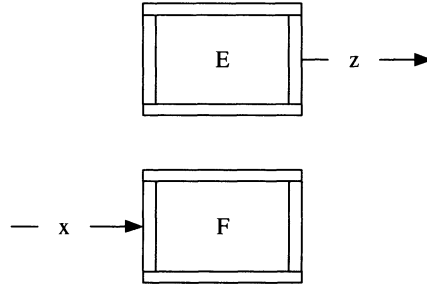
Fig. 4.5. Processes with no input or output

    process E() z: To_1
    pre P
    post Q(z)
    end_process

Since there is no input data flow to this process, no input data flow variable is given in the parentheses; the precondition should involve no input variable (usually given as **true** unless the process accesses some data stores), and the output variable z should not be defined based on any input variable. For example, we can specifically define process E as follows:

    process E() z: **nat0**
    pre **true**
    post z > 10
    end_process

The output variable z is defined independently of any other variables in the postcondition. According to this specification, a natural number greater than 10 is produced after the execution of process E. Compared with the relations between output variables and input variables given in the previously introduced process specifications, the relation z >10 implies a nondeterministic output: any natural number greater than 10 could be the output.

Process F in Figure 4.5 has no output data flow, but has an input data flow. When the process is enabled by the input data flow x, it always starts execution immediately. The effect of the execution is to consume the input data flow. As this kind of process generates no output, we call it a *sink process* or a *sink*. The formal specification of such a sink process has the form

    process F(x: Ti_1)
    pre P(x)
    post Q
    end_process
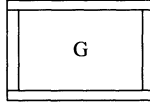
Fig. 4.6. An illegal process with no input and output
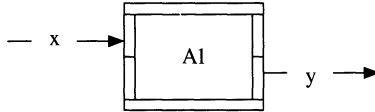
Fig. 4.7. A process with empty input port and output port

The precondition may involve the input variable x, but no output variable should occur in the postcondition. In this case, the postcondition is usually specified as **true** unless the process accesses a data store (see details of store access by processes in Section 4.5). A specific example given below can help illustrate the principle of specifying a sink process.

> **process** F(x: **nat0**)
> **pre** x > 5
> **post true**
> **end_process**

Note that processes with neither input data flow nor output data flow are illegal processes, since they provide no useful functions. For example, the process G in Figure 4.6 is illegal.

Sometimes we may need an *empty input port* and/or an *empty output port*, together with other non-empty ports. An empty port connects to no data flows. The process A1 in Figure 4.7 includes an empty input port and an output port. The process may generate y based on input x or independently of any input data flow. It may also just consume x and produce no output data flow. However, the situation of producing no output data flow independently of any input data flow is definitely disallowed, as it denotes the same situation as process G given in Figure 4.6. As an example, process A1 can be specified as follows:

> **process** A1(x: **int** |) | y: **int**
> **pre true**
> **post** x > 0 **and** y = x + 1 **or**
>   x <= 0 **and** y = x - 1 **or**
>   x = **nil and** y = 0
> **end_process**
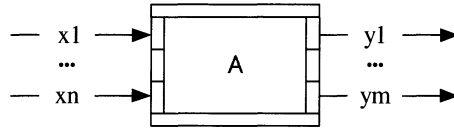
Fig. 4.8. The general structure of process

The declaration of input variables consists of two parts, separated by the symbol |. The part on the left hand side of | is the declaration of input variable x, while the part on the right hand side is an empty space denoting the *dummy* input port. Similarly, we use the same format to express the combination of dummy output port and other output variables. Process A1 requires that when x is available and greater than 0, y should be produced and be equal to x + 1; when x is available and less than or equal to zero, y should be generated and be equal to x - 1. Otherwise, if x is unavailable, y will be generated with the value zero. Since processes with this kind of structure likely lead to a complicated specification, they should be avoided if possible.

In summary of all the possible forms described so far, process A in Figure 4.8 shows the general structure of a process. Each xi (i=1...n) or yj (j=1...m) denotes a group of data flows. When all of the data flows connected to one port are available, we say that this *port is available*. This concept applies to both input ports and output ports. The meaning of this process is interpreted as the following sequence of actions:

1. when one of the input ports is available, the process A is enabled, and will not be executed until all the output ports become unavailable.
2. the execution of the process consumes all the input data flows connected to the available input port for activating the execution, and makes exactly one of the output ports available (and therefore all the output data flows connected to this port become available).

Note that a port is available if and only if all the data flows connected to it are available. In other words, if any single data flow of a port is unavailable, the entire port will be unavailable.

The formal specification of process A in Figure 4.8 is abstracted as

> **process** A(x1_dec | x2_dec | ... | xn_dec)
>   y1_dec | y2_dec | ... | ym_dec
> **pre**   P(x1, x2, ..., xn)
> **post**   Q(x1, x2, ..., xn, y1, y2, ..., ym)
> **end_process**

Each xi_dec (i =1...n) is a sequence of input variable declarations separated by comma, such as
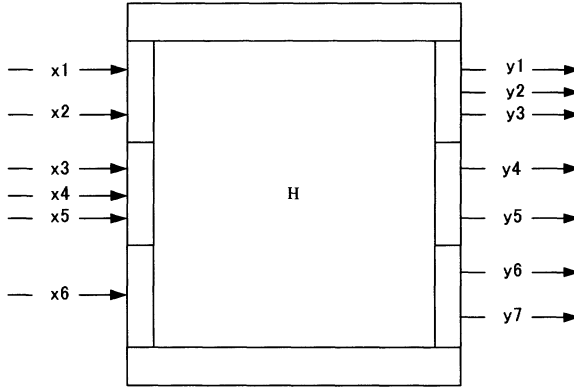
Fig. 4.9. An example of process with multiple ports

xi_1: Ti_1, xi_2: Ti_2, ..., xi_n: Ti_n

where xi_1, xi_2, ..., xi_n are the data flow variables connecting to input port xi, and Ti_1, Ti_2, ..., Ti_n are their types, respectively. Likewise, each yj_dec (j = 1...m) is a sequence of output variable declarations with a form similar to xi_dec.

For example, Figure 4.9 gives a process with multiple ports. The outline of its formal specification is

process A(x1: Ti_1, x2: Ti_2 | x3: Ti_3, x4: Ti_4, x5: Ti_5 | x6: Ti_6)
        y1: To_1, y2: To_2, y3: To_3 | y4: To_4, y5: To_5 |
        y6: To_6, y7: To_7
pre   P(x1, x2, x3, x4, x5, x6)
post   Q(x1,x2, x3, x4, x5, x6, y1, y2, y3, y4, y5, y6, y7)
end_process

where Ti_1, Ti_2, ..., Ti_6 are types for input variables x1, x2, ..., x6, and To_1, To_2, ..., To_7 are types for output variables y1, y2, ..., y7, respectively.

To avoid confusion in the formal specification of a process, a syntactical rule must be applied when a process is drawn: all the input or output data flow variables must be different from each other. Thus, the process given in Figure 4.10 is an illegal process, as the two input data flows are labeled with the same name x.

4.4 Data Flows

A data flow represents a *data transmission* from one process to another, as we have understood, more or less, from the description of the previous section.
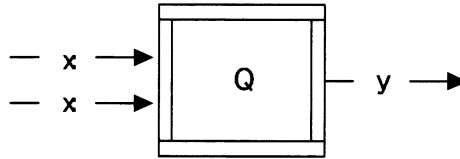
Fig. 4.10. An illegal process
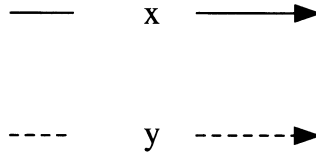
Fig. 4.11. Two kinds of data flows

A data flow has a name, denoted by an identifier, and indicates the direction in which the data are transmitted. A legal identifier is formed by an English letter followed by numbers, letters, or their combinations. For the sake of readability, the underscore symbol "_ " may also be used in identifiers.

There are two kinds of data flows. One is known as *active data flow*, and the other is *control data flow*. Figure 4.11 shows the graphical representations of these two kinds of data flows. An active data flow is denoted by a solid directed line like data flow x, while a control data flow is denoted by a dashed directed line like y. The primary function of an active data flow is to transmit the actual data that are expected to be used by another process, whereas a control data flow transmits "special data" that will not be used by another process during its execution, but its availability can enable the process (thereby playing the role of controlling the process execution). The control data flow is usually used when describing the requirement that a process be executed after the execution of its preceding process without the need for receiving any useful data flow. For example, Figure 4.12 shows that process B must be executed after process A, without the need for any "useful" data (but for a "signal" y supplied by process A), whereas process C, which must also be executed after process A, needs the actual data z supplied by process A for generating its output data flow w.

In general, the graphical symbol of a data flow given in Figure 4.11 denotes a *data flow variable*, not a specific value like in the classical data flow diagrams. When a specific value is bound to this variable, the data flow through this variable is said to become available and the variable is said to be *defined*; otherwise, the variable is said to be *undefined*. Since we use a data flow identifier, say x, as a variable and a specific data flow, alternately from time to time, for the sake of simplicity we use the following phrases equivalently:
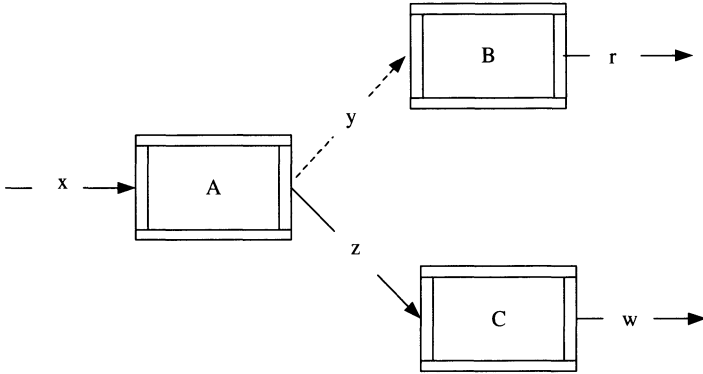
Fig. 4.12. An example of using active and control data flows

- data flow x is available.
- data flow variable x is defined.

Likewise for the following notions:

- data flow x is unavailable.
- data flow variable x is undefined.

Thus, we treat x as a data flow variable whenever we talk about whether it is defined or undefined, and as a data flow whenever we talk about whether it is available or unavailable.

Every data flow variable must be declared with a type, such as **nat**, **int**, **real**, so the binding of values to a variable must be restricted to its type. That is, when the value of its type is bound to the variable, we say this variable is defined; otherwise, it is undefined. Formally,

Definition 3. *Let x be a data flow variable of type T. x*
is defined if a value of T
is bound to x. Otherwise, x
is undefined.

This concept is similar to a water pipe system. A data flow variable is like a water pipe and a specific data flow is like water going through the pipe, as shown in Figure 4.13. When water comes in the pipe, the pipe is occupied (defined); when water flows out, the pipe becomes empty (undefined).

We use **bound(x)** to mean that variable x is defined or data through x is available. Formally, predicate **bound** is defined as follows:

bound : X −> **bool**
bound(x) = **true** if x is defined, and **false** if x is undefined

where X is a set of variables.
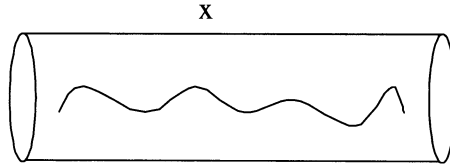
Fig. 4.13. An illustration of data flow variable and data flow

As we have seen in section 4.3, a declaration of data flow variables uses the form:

v_1, v_2, ..., v_n: T

where v_i (i = 1...n and n ≥ 1) are variables and T is a type. There are several kinds of types available in SOFL, and they will be introduced in later chapters. But for now, we use only the basic types given in discussions of the fundamental concepts at the beginning of Section 3.1 of Chapter 3.

Every control data flow variable must be declared with a special type known as *signal*, which is denoted by the symbol **sign**. This type contains only one value denoted by the exclamation mark !, serving as a signal to make the related control data flow variable defined. That is, a data flow variable of type **sign** is defined if it is bound to the value !; otherwise, the variable is undefined. Formally,

sign = { ! }

There is no operator on this type. It is worthy of notice that no active data flow variable can be declared with type **sign**.

The availability of control data flows can be represented in different ways. Let x be a control data flow variable, the following three expressions mean the same thing: variable x is defined or data flow x is available.

- **bound**(x)
- x <> **nil**
- x = !

For the sake of readability of process specifications, the third expression is not used in this book unless it is really necessary.

4.5 Data Stores

A *data store*, or *store* for short, is a variable that holds data in rest. In contrast with data flows, stores do not actively transmit data to any process; rather
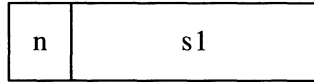
Fig. 4.14. A data store

Fig. 4.15. Different stores sharing the same name

they hold data that is always ready to supply to any process when requested. In fact, the notion of store is not unfamiliar to us, since it is available almost everywhere in our daily life. For example, a library is a store of books; a warehouse is a store of products; a child's money box is a store of cashes and coins; and so on. In a program system, a file or database can be perceived as a store. But in SOFL specifications, stores are treated more generally: they are just normal variables holding values ready for use by processes.

A store has a name and number for reference by people who are involved in the building of the specification. Figure 4.14 shows the general structure of the graphical representation of a store. The store is named s1, and identifiered by number n. As there may be many people working on the same large CDFD and the same name needs to be given to different stores, these stores can be distinguished from each other by different numbers. Figure 4.15 shows two different stores with the same name; they are different because they have different numbers. Thus, we can use the identifier my_file_1 to denote the store on the left hand side and use my_file_2 to denote the store on the right hand side in the textual formal specification of the associated module. This point will be explained in more detail later in this section.

A store can be connected, by directed lines, *only* to processes. It does not make any sense to connect a store to a data flow because a data flow has no role of making requests for data; it can only transmit data from one process to another.

There are two ways to connect a store to a process, each denoting a different kind of access to the data in the store by the process: *read* and *write*. Figure 4.16 illustrates the different connections between processes and stores. The connection between store s1 and process A on the left hand side represents a read from the store by the process during its execution. That is, process A only uses the data of store s1 in producing its output data flow y1, and s1 stays unchanged before and after the execution of the process. The connection between store s2 and process B given on the right hand side represents a writing to or updating s2 by B. This does not exclude the case of reading data from s2, but must ensure that writing to the store is definitely involved.
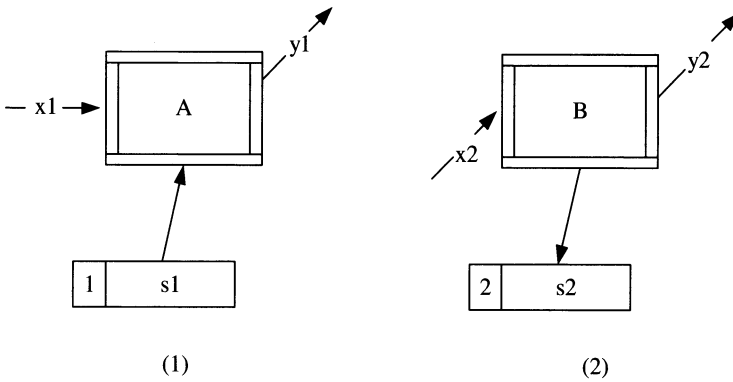
Fig. 4.16. Connections between processes and stores

A writing to the store may include the cases of updating part of the data of s2 or completely replacing the current data of s2 with new data. These two kinds of connections provide convenient ways to model communications among processes through stores.

Note that the directed lines connecting stores and processes must not be drawn from or to the input ports and output ports because they are reserved for connecting input data flows and output data flows. However, there is no restriction on where the connection lines can be drawn from or to stores: you can draw a connection line from or to anywhere on any edge of a store.

We treat a store connected to a process as an *external variable* of the process, which is, in fact, a state variable of the associated module whose CDFD contains the store. We will discuss more about the state of modules in the next section, but for now let us focus on the connection between stores and processes.

When writing a formal specification of a process accessing a store, the method of access must be properly indicated by using the keywords **rd** or **wr**. **rd** is an abbreviation for "read", and **wr** is an abbreviation for "write". A store variable declared as a **rd** external variable of a process means that the value, or part of the value, of the variable will possibly be read by the process, but not be updated during the execution of the process. A store variable declared as a **wr** external variable means that the variable will possibly be updated during the execution of the process, and it *does not* eliminate the possibility of the process reading from the variable. For example, process A in Figure 4.16 can be specified as

```
process A(x1: int) y1: int
ext rd s1: int
pre  x1 > 0 and s1 > x1
post y1 = s1 - x1
end_process
```

where the keyword **ext** is an indication of the external variable declarations. Each declaration has the form

   access variable: type

where *access* is one of the keywords **rd** and **wr**; *variable* is the variable to be declared like s1; and *type* is a type, the values of which can be bound to the variable. Note that *type* can be omitted if the corresponding store variable is declared as a state variable in the **var** section of the module; see details of the **var** section in Section 4.13.

   The specification of process A describes a subtraction: if the input data flow x1 is greater than zero and is less than the store value s1, then the subtraction of x1 from s1 will be bound to y1 as the output; otherwise, anything may happen. Since external variable s1 is not updated during the execution of process A, the value of s1 before and after the process is the same. Therefore, s1 occurring in both the pre and postcondition denotes the same variable with the same value before and after process A.

   Process B writes data to store s2, so it can be specified as follows:

   process B(x2: **int**) y2: **int**
   ext wr s2: **int**
   pre x2 > 0
   post y2 = ~s2 + x2 **and** s2 = ~s2 - x2
   end_process

This process produces output y2, which is equal to the addition of input x2 and the initial value of store s2, and updates store s2 by subtracting x2 from the initial value of s2. The external variable s2 is used in the postcondition as a two state variable: *initial external variable* and *final external variable*. The initial external variable, denoted by the decorated identifier ~s2 with the mark tilde placed before the identifier, represents the value of external variable s2 before the process, while the final external variable, denoted by the same identifier s2, represents the value of the external variable after the process.

   Note that we must not omit the tilde mark ~ in the equation s2 = ~s2 - x2 to write it as s2 = s2 - x2 in the postcondition, because it will not define the s2 properly as a result of the process. Actually, the omission of the tilde mark in this case converts the postcondition into a contradiction, which cannot be satisfied by any possible values of s2 after process B.

   Furthermore, the same store can be connected to multiple processes, and a process can be connected to multiple stores as well. The connections between a store to multiple processes mean that the store will be accessed from the processes in specific ways, indicated by the type of the connections (i.e., read or write), during their executions. The connections from a process to many different stores mean that the process will access those stores during its executions. For example, Figure 4.17 shows that process A reads data from store
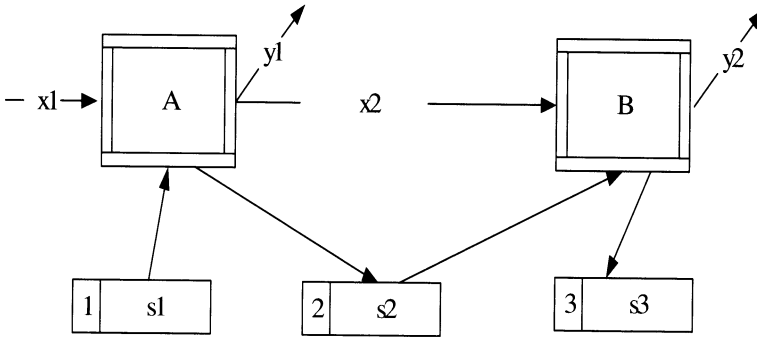
Fig. 4.17. Multiple connections between processes and stores

s1 and updates store s2, while process B reads data from store s2 and writes
to store s3. Let us take process A as an example to show how to write a formal
specification involving multiple external variables representing stores.

```
process A(x1: int) y1: int
ext rd s1: int
    wr s2: int
pre  P(x1, s1)
post Q(x1, y1, s1, ~s2, s2)
end_process
```

In the **ext** part, s1 is declared as a **rd** variable and s2 as a **wr** variable.

The general structure of a process specification considering all the possible
situations discussed so far is summarized as:

```
process A(x_1: Ti_1 | x_2: Ti_2 | ... | x_n: Ti_n)
         y_1: To_1| y_2: To_2 | ... | y_m: To_m
ext acc_1 z_1: Te_1
    acc_2 z_2: Te_2
        ...
    acc_q z_q: Te_q
pre     P(x_1, x_2, ..., x_n, z_1, z_2, ..., z_q)
post    Q(x_1, x_2, ..., x_n, y_1, y_2, ..., y_m,
          ~z_1, ~z_2, ..., ~z_q, z_1, z_2, ..., z_q)
end_process
```

Each acc_i (i=1...q) is one of the keywords **rd** and **wr**. The precondition may
involve all the input variables and external variables, depending on the type
of their access control. The postcondition may involve all the input variables,
output variables, initial external variables, and final external variables. Note

that each declaration given in this general form, such as x_1: Ti_1, can be a grouped declaration like:

x_11: Ti_11, x_12: Ti_12, ..., x_1d: Ti_1d

The specification of the process imposes a proof obligation for verifying its implementation: for any input satisfying the precondition before the execution of the process, the output of the process must satisfy the postcondition. Formally,

forall[x_1: Ti_1, ..., x_n: Ti_n, ~z_1: Te_1, ..., ~z_q: Te_q] |
  ((P(x_1, **nil**, ..., **nil**, ~z_1, ..., ~z_q) =>
    (exists[y_1: To_1, ..., y_m: To_m, z_1: Te_1, ..., z_q: Te_q] |
      Q(x_1, **nil**, ..., **nil**, y_1, **nil**, ..., **nil**, ~z_1, ..., ~z_q, z_1, ..., z_q) **or**
      Q(x_1, **nil**, ..., **nil**, **nil**, y_2, ..., **nil**, ~z_1, ..., ~z_q, z_1, ..., z_q) **or**
            ...   **or**
      Q(x_1, **nil**, ..., **nil**, **nil**, **nil**, ..., y_m, ~z_1, ..., ~z_q, z_1, ..., z_q)
)
) **or**
       ...
     or
    (P(**nil**, **nil**, ..., x_n, ~z_1, ..., ~z_q) =>
     (exists[y_1: To_1, ..., y_m: To_m, z_1: Te_1, ..., z_q: Te_q] |
      Q(**nil**, **nil**, ..., x_n, y_1, **nil**, ..., **nil**, ~z_1, ..., ~z_q, z_1, ..., z_q) **or**
      Q(**nil**, **nil**, ..., x_n, **nil**, y_2, ..., **nil**, ~z_1, ..., ~z_q, z_1, ..., z_q) **or**
            ...   **or**
      Q(**nil**, **nil**, ..., x_n, **nil**, **nil**, ..., y_m, ~z_1, ..., ~z_q, z_1, ..., z_q)
)
)
)

For the sake of simplicity, a grouped expression for bound variable constraints is adopted. For example, suppose the declaration of x1 is interpreted as the one given previously; then x1: Ti_1 is equivalent to the following sequence of constraints:

x_11: Ti_11, x_12: Ti_12, ..., x_1d: Ti_1d

Likewise, for other bound variable constraints, such as x2: Ti_2, y1: To_1, and so on.

According to this proof obligation, for any group of input data flows connected to one port and initial values of the external variables of the process, if the precondition holds before the process, then there must exist a group of output data flows connected to one output port and final values of the external variables such that the postcondition holds. However, if the precondition
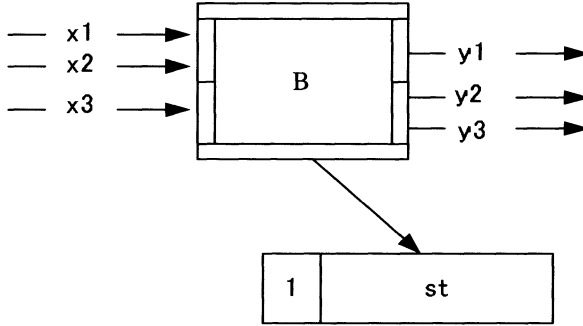
Fig. 4.18. A simple process

is false, then the postcondition can be either true or false, indicating that the specification is not responsible for such a situation. It is the operation environment that needs to ensure the truth of the precondition of the process before its execution.

A simpler case can help illustrate the proof obligation of a process. Consider the process B in Figure 4.18 as an example. The specification of process B is

> **process** B(x1, x2: **int** | x3: **int**) y1: **int** | y2, y3: **int**
> **ext wr** st: **int**
> **pre** x1 > 0 **and** x2 > 0 **or** x3 > 0
> **post** y1 > x1 + x2 + ˜st **and** st = ˜st - (x1 + x2) **or**
>   y2 + y3 >= x3 + ˜st **and** st = ˜st + x3
> **end_process**

Applying the general predicate expression of the proof obligation given previously, we derive the proof obligation of this process:

> **forall**[x1, x2: **int**, x3: **int**, ˜st: **int**] |
>   ((x1 > 0 **and** x2 > 0 **or** **nil** > 0 =>
>   (**exists**[y1: **int**, st: **int**] |
>   y1 > x1 + x2 + ˜st **and** st = ˜st - (x1 + x2) **or**
>   **nil** + **nil** >= **nil** + ˜st **and** st = ˜st + **nil**
>   )
>   ) **or**
>   (**nil** > 0 **and** **nil** > 0 **or** x3 > 0 =>
>   (**exists**[y2, y3: **int**, st: **int**] |
>   **nil** > **nil** + **nil** + ˜st **and** st = ˜st - (**nil** + **nil**) **or**
>   y2 + y3 >= x3 + ˜st **and** st = ˜st + x3
>   )
>   )
>   )

We can simplify further this expression by applying the rules that any arithmetic operation involving the "undefined" (i.e., **nil**) as an operand yields **nil**. Thus we can derive the following simplified expression:

forall[x1, x2: **int**, x3: **int**, ˜st: **int**] |
    ((x1 > 0 **and** x2 > 0 =>
     (**exists**[y1: **int**, st: **int**] |
       y1 > x1 + x2 + ˜st **and** st = ˜st - (x1 + x2)
)
) **or**
    (x3 > 0 =>
     (**exists**[y2, y3: **int**, st: **int**] |
       y2 + y3 >= x3 + ˜st **and** st = ˜st + x3
)
)
)

The proof obligation will not be used until we discuss strategies for specification construction, reviews, and testing of specifications in Chapters 15, 17, and 18, respectively. The present concern is how we should understand a process. In other words, what is the semantics of a process. This concept is essential in understanding the function of processes and in choosing implementation strategies for processes.

A process describes a *relation* between *initial states* and *final states* of the process. A relation is a collection of pairs, such as R = {(x_1, y_1), (x_2, y_2), (x_3, y_3)}. Two elements are said to have relation R if and only if the pair composed of them belongs to R. For example, x_1 and y_1 have relation R, usually represented by x_1 R y_1. An initial state is the collection of all the input data flow variables and the initial external variables, together with their values, while a final state is the set of all the input data flow variables, initial external variables, output data flow variables, and final external variables, together with their values. Thus, a formal definition of the semantics of process A is

Sem(A) = {(s_0, s_1) | **pre_**A(s_0) **and post_**A(s_0, s_1)}

where Sem(A) denotes the semantics of process A; s_0 and s_1 are the initial and final states, respectively; and **pre_**A(s_0) and **post_**A(s_0, s_1) denote the pre and postconditions of process A, respectively.

It is worth noting that process A actually associates only those initial states satisfying the precondition to the final states meeting the postcondition. All the initial states that do not satisfy the precondition have no precisely defined final states to be associated with. In other words, the final states corresponding to the initial states not satisfying the precondition are *uncontrollable* by the postcondition.

4.6 Convention for Names

The names of processes, data flows, and stores are denoted by identifiers. An identifier is a string of English letters, digits, and the underscore mark, but the first character must be a letter. An identifier is case sensitive, so Student_1 is different from student_1.

Syntactically, it is the convention in SOFL that names of processes are usually written with an upper case letter for the first character of each English word and lower case letters for the rest of characters, whereas names of data flows and stores are usually written using lower case letters for all the characters. If more than one English word is involved in a name, those words are separated by the underscore mark. Digits can be freely combined with letters and the underscore mark in names. For example, Receive_Command and Check_Password in Figure 4.1 denote two processes, respectively, while card_id, pass, and w_draw are used to name data flows.

To provide good readability of CDFDs, the names of processes, data flows, and stores should be given appropriately so that they convey the potential meaning of the corresponding processes, data flows, or stores. Usually, process names should indicate the potential functionality or behavior, such as Receive_Command and Check_Password, while the names of data flows and data stores should indicate the nature of the data they are carrying or holding. For example, card_id, pass, and w_draw are sensible names for data flows, and account_file is an acceptable name for the store in Figure 4.1.

4.7 Conditional Structures

In addition to processes, data flows, and stores, other components, known as *structures*, are also provided in SOFL for the construction of complex CDFDs. In this section, conditional structures are introduced, and other structures will be discussed in the following sections.

There are three kinds of conditional structures: *single condition structure*, *binary condition structure*, and *multiple condition structure*. These structures correspond respectively to the *if-then-* statement, *if-then-else-* statement, and *case* statement available in many programming languages like Pascal, but with some differences in both syntax and semantics. Each conditional structure consists of a node, denoted by a *diamond* or *box*, input data flow, and output data flows. Figure 4.19 shows the graphical representations of these three structures.

The data flow on the left hand side of each node is known as *input data flow*, and the data flows on the right hand side of the node are called *output data flows*.

The single condition structure means that when input data flow x is available and satisfies condition C(x), then the data flow will be passed to variable

Single condition structure

Binary condition structure

Multiple condition structure
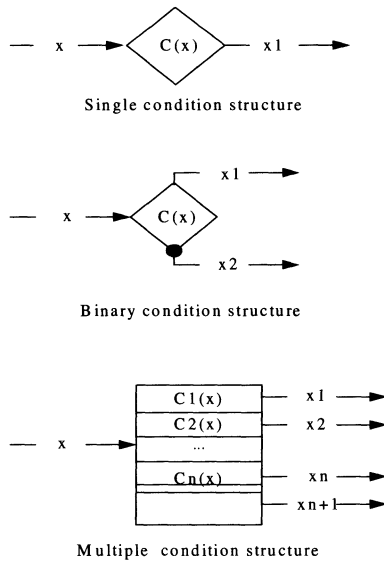
Fig. 4.19. Conditional structures

x1. In fact, the single condition structure is like a process with the following operational semantics:

1. if x is available and satisfies condition C(x), x1 is generated to have the same value as x, and x is consumed.
2. if x is available and does not satisfy condition C(x), x is just consumed, without generating x1.

The binary condition structure allows a binary choice in moving data items between processes, and its semantics is interpreted as follows:

1. if data flow x is available and satisfies condition C(x), then data flow x1 with the same value as x will be made available.
2. otherwise, if C(x) evaluates to false, then data flow x2 with the same value as x will be made available.
3. in either case above, the input data flow x will be consumed.
4. if C(x) is "undefined," x is just consumed, without producing any of x1 and x2.

Note that the small black circle marks the branch when C(x) evaluates to false.

The multiple condition structure allows only one data flow to be generated based on one input data flow under the multiple conditions. Specifically, the multiple condition structure in Figure 4.19 means that

1. if x is available and satisfies condition $C_i(x)$ ($i=1...n$), the corresponding data flow xi with the same value as x is made available.
2. otherwise, if $C_i(x)$ evaluates to false, then $C_{i+1}(x)$ will be tested, and such tests go on until one of the $C_i(x)$ ($i=1...n$) evaluates to true.
3. however, if none of $C_1(x)$, $C_2(x)$, ..., $C_n(x)$ are satisfied by x, then xn+1 with the same value as x is made available as default.
4. in any case above, the generation of the output data flow results in the consumption of x.

Note that the multiple condition structure is similar to a case statement in the conventional programming language like Pascal. That is, the conditions of $C_1(x)$, $C_2(x)$, ..., $C_n(x)$ are tested in turn, and once one of them evaluates to true, the corresponding output data flow will be generated. However, if none of the given conditions is true, the default data flow will be generated. Note that the default data flow must always be provided as a requirement of the multiple condition structure. In this way, we ensure that the input data flow is always transformed to an output data flow of the structure.

In Figure 4.19 we use different variables to name the output data flows, although they have the same value as the input data flow. However, this is not required by SOFL syntax. As long as there is no confusion in distinguishing data flows, the same identifier can be used to denote both the input and the output data flows of a conditional structure. For example, we can rename all the output data flows in the multiple condition structure as x; this will not cause confusion in specification because they are distinguished by their drawing positions.

Usually the output data flows are connected to different processes: they are treated as the input data flows of those processes. However, it is not impossible to have more than one different output data flow of a conditional structure connecting to the same process. If this is the case, we must make sure that all of those data flows are named differently, as required by the syntax of process.

4.8 Merging and Separating Structures

The conditional structures given in Figure 4.19 have a common feature: they all have a single input data flow. That is, the condition $C(x)$ can only apply to the single input data flow x. However, sometimes we may need to decide where to transfer more than one data flows based on a condition involving all of those data flows. Such a case obviously cannot be described by using the conditional structures introduced so far. One solution to this problem is to model this case as a process: the multiple data flows are taken by the process and the output data flows with the same values are generated to transmit data to the expected destinations. However, the graphical representation of this process may not be straightforward in conveying sufficient information about such a conditional decision, because without understanding its formal

Merging structure

Separating structure
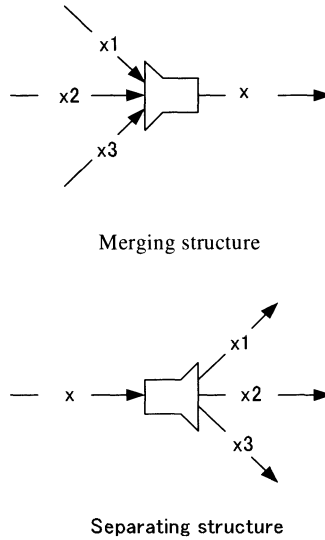
Fig. 4.20. Merging and separating structures

specification written in the textual format, it is difficult to figure out the behavior of this process. Since this kind of conditional decision may often be used in complex CDFDs, SOFL provides a pair of straightforward *merging* and *separating structures* to resolve this problem.

Figure 4.20 shows the graphical representations of the merging and separating structures. Like the conditional structures, both the merging and separating structures are composed of three parts: node, input data flows, and output data flows. The merging structure composes input data flows x1, x2, and x3 into a single composite data flow x. That is, x is formed as a composite object with the three fields x1, x2, and x3 as its components. The separating structure is opposite to the merging structure: it breaks up the composite data flow x into its components x1, x2, and x3. In fact, most of the time the merging structure and separating structure are used as a pair to describe the control of data flows.

Figure 4.21 gives an example illustrating the use of the merging and separating structures. Data flows x1, x2, and x3 generated by processes A1, A2, and A3, respectively, are merged into the single composite data flow x. If x satisfies condition C(x), then x is diverted to the upper level data flow of the binary condition structure, and then divided into the three data flows x1, x2, and x3 by the related separating node. These data flows are the same as the input data flows x1, x2, and x3 of the merging node in both type and value. On the other hand, if x fails to meet condition C(x), then x is diverted to the lower level data flow of the binary condition structure, and then separated into the original component data flows x1, x2, and x3.
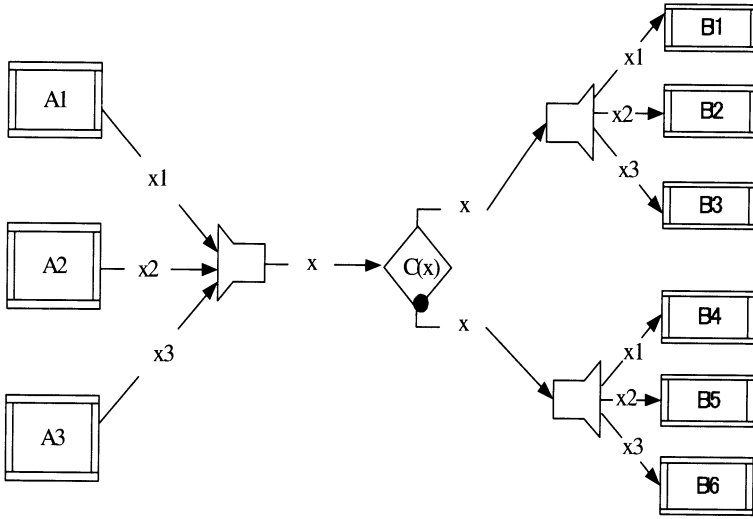
Fig. 4.21. An example of merging and separating structures

As we can see from this example, using the merging and separating structures can help us achieve straightforward CDFD, but it may also require us to draw more data flows compared to the use of a process. For example, the CDFD given in Figure 4.21 describes a specification equivalent to the CDFD shown in Figure 4.22, provided that process A is specified as follows:

process A(x1: Ti_1, x2: Ti_2, x3: Ti_3) y1: Ti_1, y2: Ti_2, y3: Ti_3 |
                            z1: Ti_1, z2: Ti_2, z3: Ti_3
  post (C(x1, x2, x3) => y1 = x1 **and** y2 = x2 **and** y3 = x3) **and**
        (**not** C(x1, x2, x3) => z1 = x1 **and** z2 = x2 **and** z3 = x3)
  end_process

where C(x1, x2, x3) is equivalent to C(x) occurring in the CDFD in Figure 4.21. They may appear to be slightly different due to the difference in the syntax between using x1, x2, and x3 as the components of a composite object and as individual variables. For example, C(x) may be something like x.x1 > x.x2 + x.x3, while C(x1, x2, x3) may be something like x1 > x2 + x3.

In the case of the CDFD in Figure 4.22, we may not have a clear idea about what the CDFD describes by just looking at the CDFD itself, without referring to the textual specification of process A, but the entire CDFD has a simple structure. On the other hand, the CDFD given in Figure 4.21 provides a straightforward picture about the behavior of the CDFD, but it has a relatively complicated structure. Which approach should be chosen is really a matter of the user's preference, although the complexity of the specific CDFD may need to be taken into account.
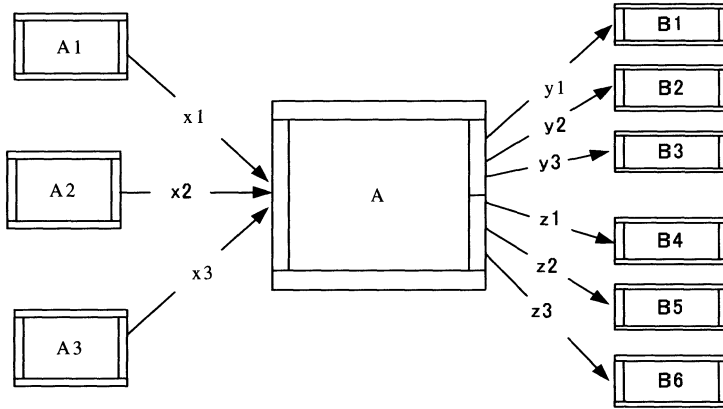
Fig. 4.22. An example of modeling the merging and separating structures using process

4.9 Diverging Structures

A diverging structure transforms an input data flow to either one of the output data flows or all of the output data flows, depending on the type of the diverging structure. There are two diverging structures: *nondeterministic structure* and *broadcasting structure*, as shown in Figure 4.23. Again, each of these structures is composed of three components: input data flow, node,
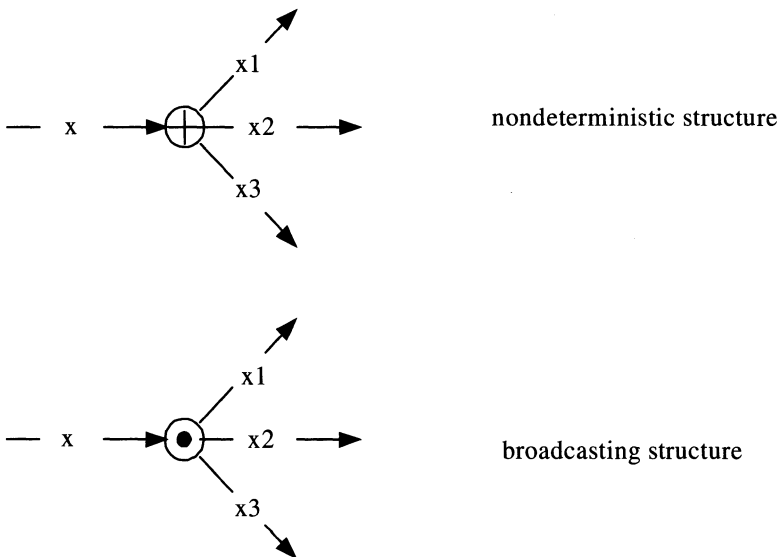
Fig. 4.23. Diverging structures
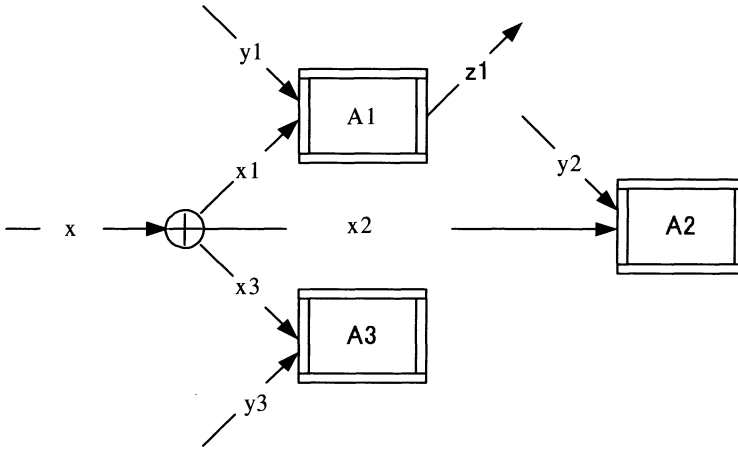
Fig. 4.24. An example of applying the nondeterministic structure

and output data flows. The node involved in the nondeterministic structure is known as *nondeterministic node*, while the node involved in the broadcasting structure is called *broadcasting node*.

In the nondeterministic structure the input data flow x is converted into exactly one of the output data flow variables x1, x2, and x3 in a nondeterministic manner. Meanwhile, the input data flow x is consumed.

The nondeterministic structure is usually used to describe the situation when a choice of the processes to which a data item flows needs to be made without the necessity of knowing exactly which one will be chosen. Let us look at the CDFD in Figure 4.24. The data in variable x can be converted to one of x1, x2, and x3, depending on which of the processes A1, A2, and A3 can be first enabled (not necessarily executed). For instance, when x is available, and data flow y1 is available before y2 and y3 become available, then x1 will be made available in order to enable process A1, and x is consumed. Once the execution of A1 terminates, data flow x1 is consumed. However, if all of y1, y2, and y3 are available before x becomes available, then any of A1, A2, and A3 can be enabled. In this case, the nondeterministic structure cannot tell which process is enabled. Such a nondeterministic situation provides a freedom for implementation: the implementer can choose any strategy appropriate to the specific application to implement this nondeterministic structure.

In the broadcasting structure in Figure 4.23, the input data flow x is transmitted to all of the output data flows x1, x2, and x3, and x is consumed after the transmission. If no confusion occurs, output data flows x1, x2, and x3 can be named the same as input data flow x. This structure is usually used when a data flow needs to be used as input by several processes. Figure 4.25 shows an example of applying the broadcasting structure in a CDFD. Data
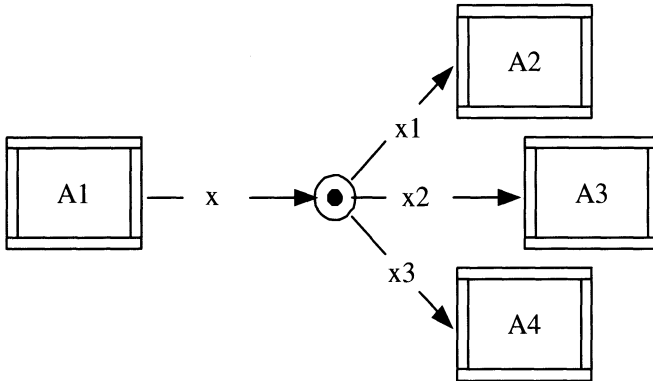
Fig. 4.25. An example of applying the broadcasting structure

flow x generated by process A is transmitted to processes A2, A3, and A4 through equivalent data flows x1, x2, and x3.

4.10 Renaming Structure

The renaming structure is intended to allow the change of data flow variables without affecting their data (values of the corresponding types). Figure 4.26 gives the graphical representation of the renaming structure.

This structure changes data flow variable x1 to y1, x2 to y2, ..., xn to yn, without changing their values. That is, each data flow yi (i=1..n) represents the same data flow as xi, but with a different variable.

The renaming structure is usually used to resolve confusion of data flow variables when an already defined process is reused in a CDFD. Assume that we want to draw a CDFD that intends to reuse the behavior of process A1 in
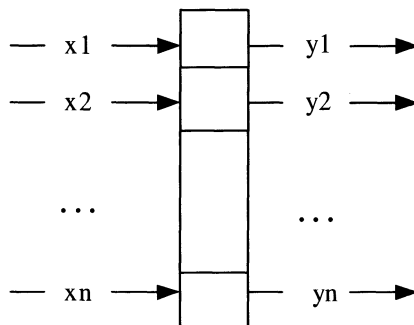
Fig. 4.26. The renaming structure
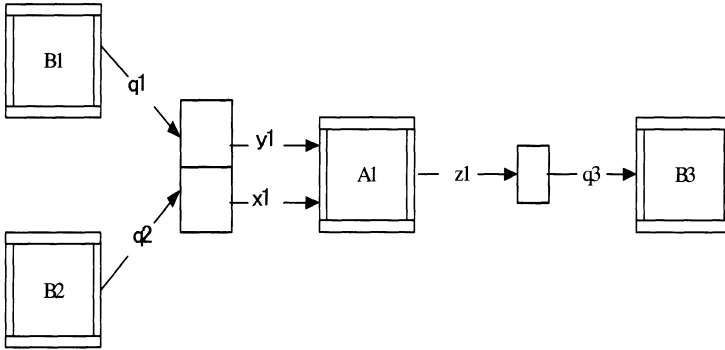
Fig. 4.27. An application of renaming structures

Figure 4.24, but not necessarily its syntactical structure (e.g., input variables). Suppose process A1 is specified as follows:

process A1(y1: int, x1: int)
pre x1 + y1 > 0
post true
end_process

Then we reuse process A1 in the CDFD given in Figure 4.27. The two renaming structures in the diagram are used to keep both the same input and the same output variables of process A1 used. Thus, the consistency between the graphical representation of process A1 and its formal specification can be sustained, which helps to improve the readability of the entire module specification.

4.11 Connecting Structures

A pair of connecting structures are available; they are used together to establish a connection of data flows in a complicated CDFD in order to reduce complexity and potential confusion of data flows. Figure 4.28 shows the two connecting structures. One is composed of the *connecting node*, denoted by a circle with a number n in it, and an input data flow, while the other consists of a connecting node and an output data flow. Note that the data flow variable x and the number n in both connecting nodes must be kept same in order to establish a consistent connection of the data flow x.

When used together in a CDFD, this pair of connecting structures indicates that input data flow x of the connecting node in Figure 4.28(a) is connected to the node in Figure 4.28(b), and further transferred to the output data flow of this node. Thus, crossing data flows, such as x1 and x2 in

(a)

(b)
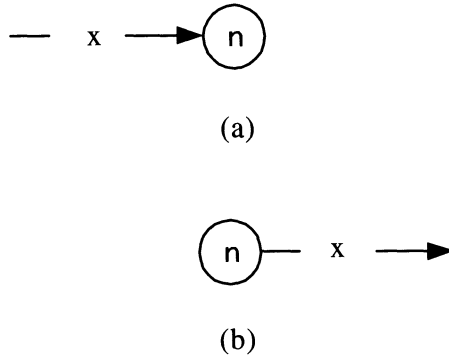
Fig. 4.28. The connecting structures
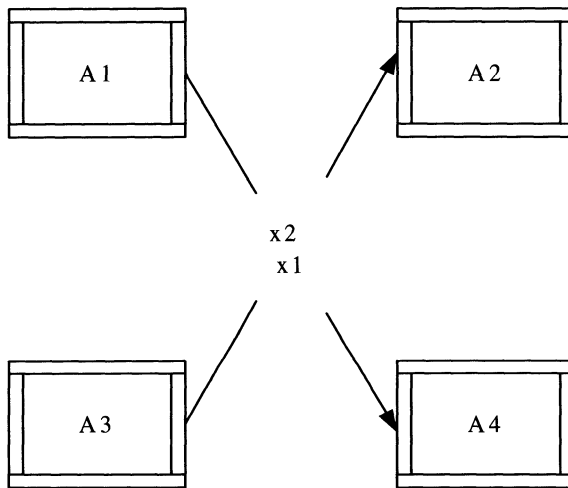
Fig. 4.29. A CDFD with crossing data flows

Figure 4.29, can be avoided by using the connecting structures. For example, the CDFD in Figure 4.29 can be changed to the one given in Figure 4.30, which resolves the confusion of data flows x1 and x2. The connecting structures are especially useful when a big CDFD is drawn on different pages.

4.12 Important Issues on CDFDs

To help build desirable CDFDs using the components introduced so far and to understand their behaviors as a whole, we need to address some fundamental issues related to both syntax and semantics of CDFDs. For example, how
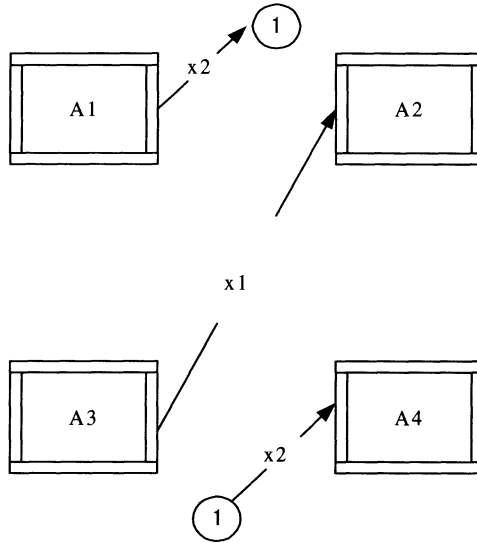
Fig. 4.30. An example of applying connecting strucutres

can a CDFD be enabled, executed, and terminated? In order to answer these important questions, it is essential to introduce necessary concepts, such as *starting process*, *starting node*, *terminating process*, and *terminating node*. In this section, we discuss these concepts in detail step by step.

4.12.1 Starting Processes

Starting processes of a CDFD serve as the starting points for enabling and executing the CDFD.

Definition 4. *A starting process of a CDFD is a process with an empty input port or an input port whose data flows are not the output data flows of any other processes and structures in the same CDFD.*

In other words, a starting process of a CDFD can be enabled without the need for executing any other processes in the same CDFD. This includes two cases: a process with an input port, whose data flows are not connected to any other processes and structures in the CDFD, and a source process (process with no input data flow). For example, in Figure 4.31, process A1, which takes input data flow x1 or x2, and A2, which takes no input data flow, are both starting processes of the CDFD.
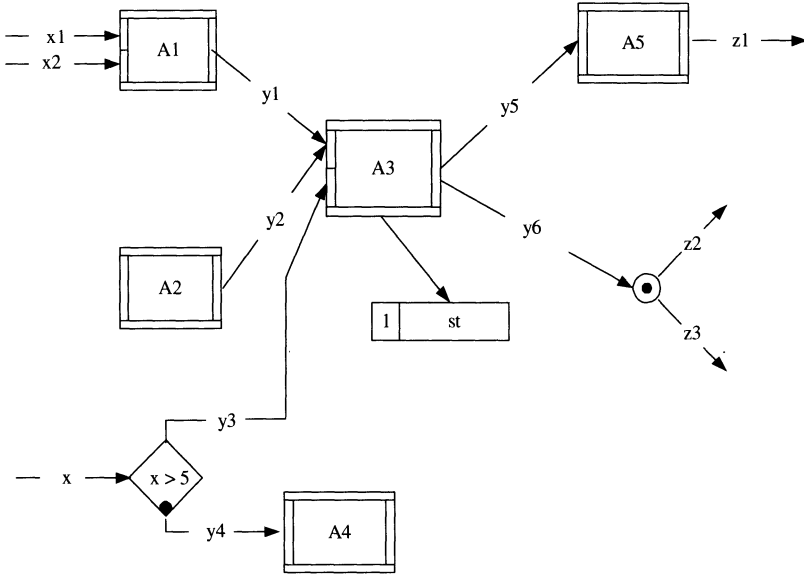
Fig. 4.31. An example of CDFD

4.12.2 Starting Nodes

A starting node is a more broad notion than starting process.

Definition 5. *A starting node is either a starting process or any of the nodes involved in the conditional structures, merging and separating structures, diverging structures, and renaming structures whose input data flows are connected to no processes and nodes in the same CDFD.*

In fact, a node involved in the structures available in SOFL can be modeled as a process. Like a process, when the input data flow of a node becomes available and all of its output data flows are unavailable, the node is enabled and executed, and then its output data flows are made available. So, it is not strange to use the term starting node to mean both starting process and node involved in those structures whose input data flow is not connected to any other processes in the same CDFD. For example, the node in the binary condition structure with input data flow x in Figure 4.31, as well as processes A1 and A2, are all starting nodes of this CDFD.

4.12.3 Terminating Processes

A terminating process of a CDFD represents a terminating point of the CDFD.

Definition 6. *A terminating process is a process with an empty output port or an output port whose data flows are not the input data flows of any other processes and structures in the same CDFD.*

According to this definition, several kinds of processes can be terminating processes. The following list gives all the possibilities:

- a process with no output data flow (empty output port).
- a process with an output port whose data flows connect to no processes or structures in the same CDFD.
- a process with one empty output port together with other non-empty output ports.

For example, process A4 and A5 in Figure 4.31 are terminating processes.

4.12.4 Terminating Nodes

Similar to a starting node, a terminating node is also a more broad notion than terminating process.

Definition 7. *A terminating node is either a terminating process or a node in the conditional structures, merging and separating structures, diverging structures, and renaming structures that has no output data flow connecting to other processes and nodes.*

For example, processes A4 and A5, as well as the node in the broadcasting structure in Figure 4.31, are all terminating nodes of the CDFD.

4.12.5 Enabling and Executing a CDFD

A CDFD can be enabled and executed to provide behaviors like a process, but the question is how the CDFD is enabled and executed, and under what condition the termination of the execution can be determined. The answers to these questions are provided by the definitions given below.

Definition 8. *A CDFD is enabled if one of its starting nodes is enabled.*

In other words, as long as one of the starting nodes of a CDFD is enabled, we say that the CDFD is enabled. As we have studied early in this chapter, enabling a process depends on the availability of its input data flows. There may be several ways to make the input data flows of a starting node available. They may be made available by some *event* that has happened outside the system under construction; for example, the arrival of a train at a railway crossing sends a signal, which can be modeled as a data flow, to activate the process that serves as a crossing controller. Sometimes, the input data flows of a starting node may be generated by the preceding processes in the high level CDFD; see more details about this issue in Chapter 5, where the hierarchical CDFDs are discussed.

Definition 9. *A CDFD starts execution if one of its input nodes starts execution.*

This means that a CDFD starts an execution by executing one of its input nodes. Each execution traverses paths from a starting node, through some intermediate nodes, down to some terminating nodes of the CDFD.

Definition 10. *An execution of a CDFD is said to be terminated if the following two conditions are satisfied.*

- *All the terminating nodes are terminated.*
- *No process in the CDFD is enabled.*

The condition that all the terminating nodes are terminated is important. However, we must understand that this condition does not necessarily require that all of the terminating nodes be actually executed, since conditional and nondeterministic nodes may be involved. For example, the execution of the CDFD in Figure 4.31 may start from the binary condition node when input data flow x is available while x1 and x2 are not available. If condition $x > 5$ is true, then data flow y3 is made available, which then enables process A3 to execute. As a result of the execution, both data flows y5 and y6 are generated. These two data flows are then taken by process A5 and the broadcasting node, both of which are terminating nodes, as input to generate their output data flows z1, z2, and z3. In this execution the terminating process A4 is not involved, and therefore it stays in the terminated state during this execution all the time. On the other hand, if condition $x > 5$ is false, then process A4 is enabled and executed, whereas the other two terminating nodes, process A5, and the node in the broadcasting structure remain terminated all the time.

The second condition given in this definition is also essential in determining the termination of a CDFD. With this condition, we can avoid mistaking the situation for termination of the CDFD: some terminating nodes are terminated while some processes in the CDFD are still executing that may eventually result in the execution of some other terminating nodes.

4.12.6 Restriction on Parallel Processes

In a CDFD, parallel execution of processes is allowed, and it can be utilized for modeling systems in several ways. Firstly, parallel processes in a CDFD can be perceived as a mechanism to describe *nondeterminism* in process executions. Let us take the CDFD in Figure 4.32 as an example to see what exactly this means. When process A1 is executed on the availability of data flow x1, the two data flows y1 and y2 are generated. Since they are the only input data flows to processes A2 and A3, respectively, A2 and A3 are enabled and can be executed concurrently. However, when this CDFD is implemented in a sequential programming language like Pascal or C, the two processes must be scheduled to perform their executions in a sequential manner, but the order of the executions does not really matter in this case because they both do not change the shared store student_files (they only read data from the store). In other words, the CDFD, as a system specification, presents a nondeterminism
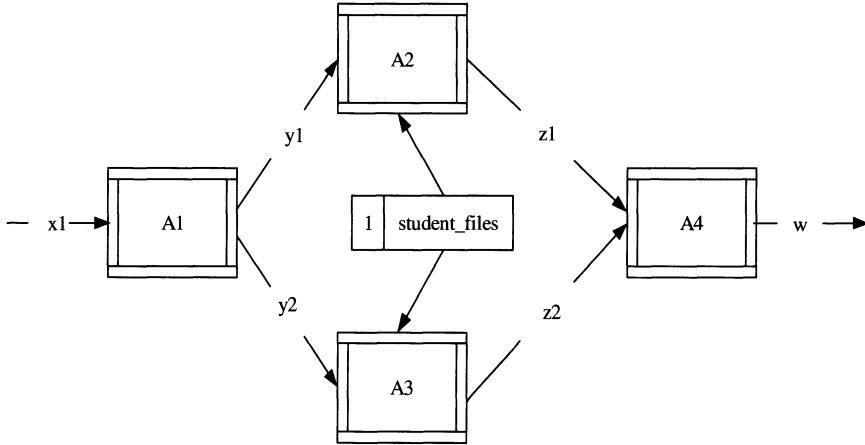
Fig. 4.32. A CDFD involving parallel processes

for its implementation using a sequential programming language. This is indeed an advantage because it allows the analyst to concentrate on "what to do" rather than "how to do it" in the early phases of software development. It also offers the programmer freedom to choose the most appropriate and efficient algorithm to implement the specification.

Secondly, parallel processes can be used to model concurrency and parallel executions in the real world systems, such as railway control systems. Thus, the CDFD models of the systems may provide the best matching between the specifications and the real systems, and therefore facilitate communications between the users and the analysts during the validation of the system specifications.

Another potential application of CDFD is to model concurrent or parallel systems, such as network computing. This kind of system may be implemented using a concurrent programming language like Java (using multi-threading). However, we must bear in mind that CDFD is not suitable for modeling systems that require communications between processes *during* their executions. The communications between two processes in CDFDs are performed by input and output data flows, as well as stores, in a sequential manner. That is, after the termination of the execution of one process, another process can possibly receive its output data flows as its input, and then executes to provide the desired behavior. If these two processes, like A2 and A3 in Figure 4.32, access the same store like student_files, the access must be controlled properly to avoid the violation of data integrity.

The condition for the restriction on accessing data stores by parallel processes is as follows:
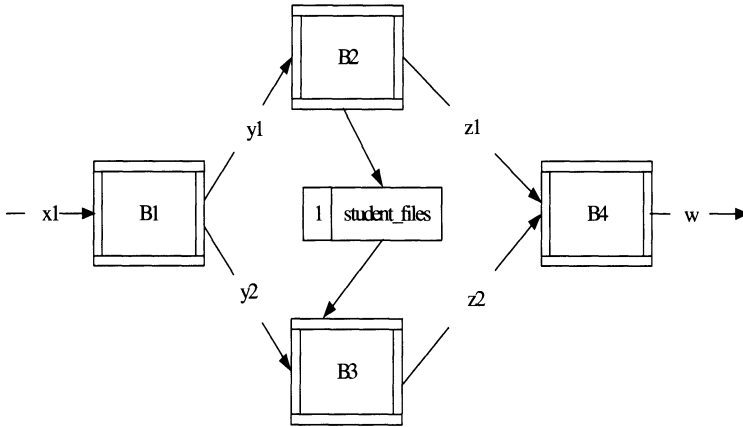
Fig. 4.33. An illegal CDFD

Condition 1 *Any two parallel processes that access the same store in a CDFD cannot write to the store during their executions.*

The reason for this restriction is because two parallel processes accessing the same store by writing or by reading and writing would possibly cause confusion in updating and reading the data of the store. Consider the CDFD in Figure 4.33 as an example. Processes B2 and B3 execute in parallel and B2 writes to the store student_files while B3 reads from the store. In this case, the order of writing to and reading from the store is nondeterministic. Therefore, the behavior of the entire CDFD may not be precisely defined.

However, if one really wants to specify that process B2 first write to store and then process B3 reads from the store, one can use a control data flow, like col in Figure 4.34, to connect processes B2 and B3. Thus, as the execution of process B3 needs the availability of data flow col generated by process B2, B2 must be executed before B3. Note that a connecting node is used to avoid crossing of data flows in the CDFD.

Note that this restriction is obviously not applicable to those parallel processes that only read from the same store, such as A2 and A3 in Figure 4.32. This is because they have no possibility of causing confusion in using the shared store.

4.12.7 Disconnected CDFDs

When building a complex system, sometimes it may be necessary to draw a disconnected CDFD, especially when such a CDFD is a decomposition of a high level process. As process decomposition will be discussed in detail in Section 5.1 of Chapter 5, let us now concentrate on the notion of disconnected CDFD rather than how they can possibly be created.
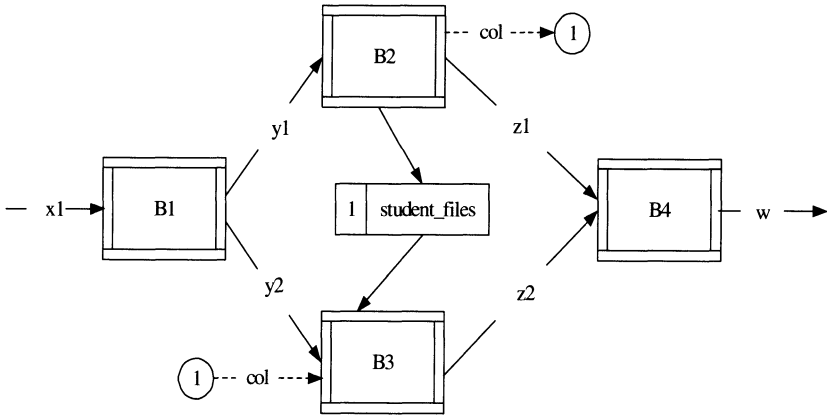
Fig. 4.34. A CDFD with no confusion
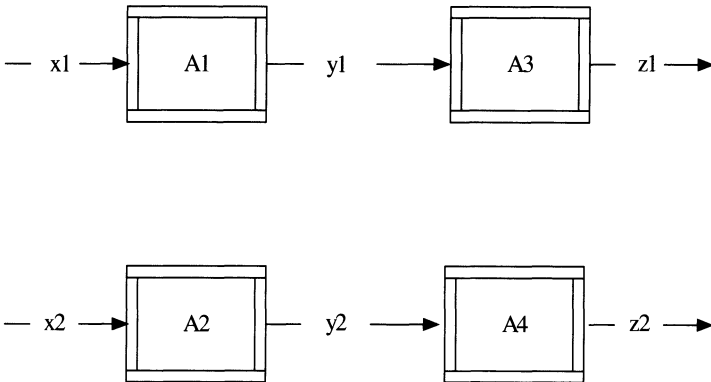
Fig. 4.35. An example of disconnected CDFD

Definition 11. *A disconnected CDFD is a disconnected graph: there exists at least one process or node, such as a conditional, diverging structure, which is not reachable through a data flow path (a sequence of data flows) from every starting process and node in the CDFD.*

In this definition, "a sequence of data flows" means an ordered data flows, disregarding their directions, For example, the CDFD given in Figure 4.35 is a disconnected CDFD because process A3 is not reachable from the starting process A2 through any data flow path and A4 is not reachable from A1 through any data flow path.

   As we will see in the next chapter, every CDFD, except the top one, is a decomposition of a high level process in an entire specification. A disconnected CDFD may be inevitable when it is derived from decomposing a high level process, with several input and/or output ports connected to different groups
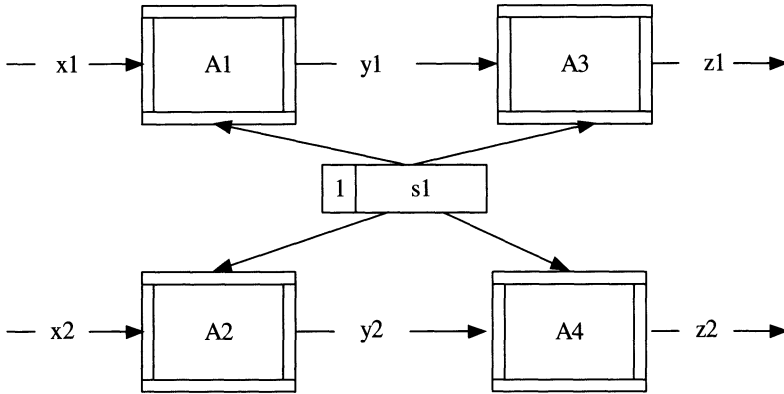
Fig. 4.36. A disconnected CDFD involving a store

of data flows, in order to keep logical coherence between the high level process and the decomposition.

There are three possibilities to start executing the CDFD in Figure 4.35. One is when data flow x1 is available, and another is when data flow x2 is available. The third possibility is when both x1 and x2 are available. It is not difficult to imagine the execution scenarios according to the rules for executing CDFDs, but it may not be straightforward to determine how to start the execution of the CDFD when both x1 and x2 are available because there are two possibilities. One possibility is to enable one of the processes A1 and A2 by the availability of data flow x1 or x2, but not both. Another possibility is that both x1 and x2 are used to execute the CDFD in parallel, which eventually leads to the generation of data flows z1 and z2. The way to execute the CDFD in this case may sound nondeterministic, but since this CDFD is usually a decomposition of a high level process in a specification, it may be defined precisely by the high level process.

Note that data stores might be involved during the execution of the processes in Figure 4.35, such as in the situation illustrated in Figure 4.36. Although this CDFD looks like a connected CDFD because of the connections between store s1 and all the processes, it is still a disconnected CDFD by definition. Note that the connections between the store and processes are not data flows; they are just indications of store accesses by the processes.

4.12.8 External Processes

When modeling a system with a CDFD, we should pay attention not only to the correctness and preciseness of the CDFD, but also to the readability of the CDFD. One way to improve the readability of a CDFD is to show explicitly the *entities* that provide input data flows to the starting nodes of the CDFD or that receive output data flows produced by terminating nodes. Such an
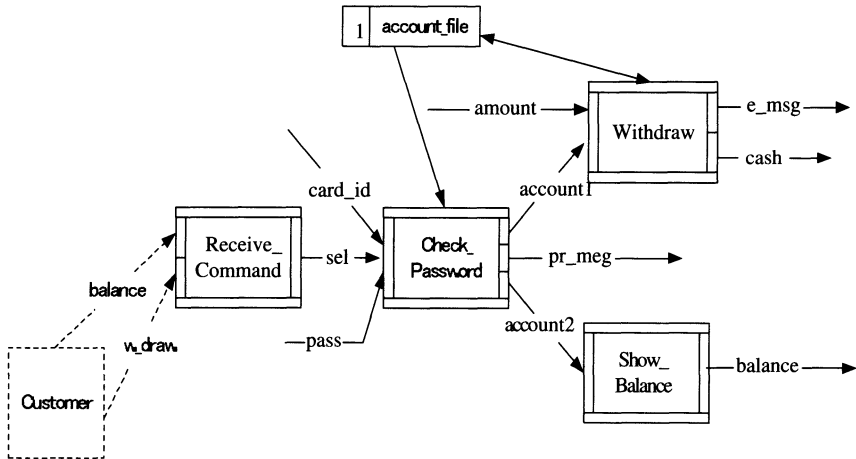
Fig. 4.37. The CDFD with an external process

entity may be a person, machine, organization, a group, or any object with the function of providing useful data information to the system concerned. Since such entities may not be suitable for being part of the system, we need to distinguish them from the normal processes used in the system. We call such entities *external processes*, because their behavior can be modeled as a process. An external process is represented graphically by a dashed-line box, such as the external process customer in the CDFD given in Figure 4.37.

Since external processes are not treated the same as normal processes in CDFDs, their syntax and semantics do not need to conform to the rules for normal processes; they are just designed to provide useful information about the system to help communication between the developer and the user via the CDFDs. For example, the external process in Figure 4.37 is named customer, because this information can help us to understand who provides the command for displaying the balance of or withdrawing cash from the account.

4.13 Associating CDFD with a Module

A graphical notation like CDFD is comprehensible, but may not be capable of defining all the components precisely for the sake of the space it occupies. As we have described previously, the components of a CDFD can be formally defined. To organize all the formal definitions related to the CDFD, the concept of a module is provided.

A module is an encapsulation of data and processes with a behavior represented by the CDFD it associates with. Generally speaking, a module has the following structure:

```
module ModuleName / ParentModuleName;
const ConstantDeclaration;
type TypeDeclaration;
var VariableDeclaration;
inv TypeandStateInvariants;
behav CDFD_no;
InitializationProcess;
Process_1;
Process_2;
      ...
Process_n;
Function_1;
Function_2;
      ...
Function_m;
end_module
```

The keyword **module** indicates the start of a module. *ModuleName* provides a distinct identity of the module in the specification. Since the module probably describes a decomposition of a high level process that is defined in another module, that module name must be provided as *ParentModuleName* in order to build an explicit connection between the high level process and its decomposition for good traceability of the entire specification. Such a traceability will facilitate reading and modifying of the specification. Since process decomposition will be introduced in detail in the next chapter; the reader does not need to pay much attention to this issue for now.

The keyword **const** starts the part for constant declarations. A constant with a special meaning may be frequently used in process specifications, but it may be subject to change when the specification is modified for whatever reason (e.g., to fit requirements changes or module version changes for different systems). *ConstantDeclaration* may consist of several declarations, separated by semicolons, as

$$ConstIdentifier\_1 = Constant\_1;$$
$$ConstIdentifier\_2 = Constant\_2;$$
$$...$$
$$ConstIdentifier\_q = Constant\_q;$$

Each *ConstIdentifier_i* ($i = 1..q$, $q \geq 1$) is a constant identifier and *Constant_i* is the value of any type available in SOFL. Each equation declares a constant identifier on the left hand side of the equality symbol $=$ as the constant on the right hand side of $=$. For example, we declare a constant identifier **age** representing the age of lawfully becoming adult in Japan as follows:

const age = 20;

Thus, `age` can be used to mean the constant 20 whenever 20 is intended to be used in process specifications. In the future, if the age of becoming adult should change to 18 for whatever reason (e.g., change of the law), there would be no need to change the constant 20 throughout the entire specification, but only to change it in the `age` declaration.

The keyword **type** starts the part of type declarations. *TypeDeclaration* is usually a list of several type declarations and its structure is similar to that of *ConstantDeclaration*, as shown below:

$$TypeIdentifier\_1 = Type\_1;$$
$$TypeIdentifier\_2 = Type\_2;$$
$$...$$
$$TypeIdentifier\_w = Type\_w;$$

TypeIdentifier_i ($i = 1..w$ and $w \geq 1$) are normal identifiers, but it is the convention to name a type identifier as a sequence of several words, if applicable, with an upper case letter for the head character of each word. For example, UniversityStudentFile is a qualified type identifier. Each *Type_i* is either another already defined type identifier or a specific type built by applying the corresponding type constructor available in SOFL. For the basic types like **nat0**, **nat**, **int**, **real**, and **string**, their type constructors are the same as their names. However, as we will see in later chapters, compound types are also available and their type constructors may vary depending on their constituent types; see the details in Chapters 8, 9, 10, and 11. An equation in the type declaration means that the type identifier on the left hand side of the symbol = is declared as the type given on the right hand side.

For example, the type identifier Address is declared as follows:

Address = **string**;

Of course, we can use type **string** itself directly in the specification rather than declaring Address. However, Address may make better sense than **string**, because a string can be used to denote many different things whereas Address is rather specific. Thus, the use of Address in process specifications may help to improve the readability of process specifications.

Sometimes there may be a need to declare a type identifier, without indicating what specific type it will represent. In this case, the keyword **given** is used to indicate that the declared type identifier represents a given type (simply a set of unknown values). For instance, a given type Employee is defined as

Employee = **given**;

Thus, Employee can only be used as a *given* type, and nothing can be said about its values.

The keyword **var** marks the start of variable declarations. In this part, all the data stores occurring in the associated CDFD of the module must be declared with some types. Again, like type declarations, the variable declaration part may include a list of individual variable declarations, separated by semicolons. The structure of *VariableDeclaration* is:

 Variable_1: *Type_1*;
 Variable_2: *Type_2;*
   ...
 Variable_u: *Type_u*;

Each *Variable_i* ($i = 1..u$) is an identifier, usually denoting a data store occurring in the associated CDFD, and *Type_i* is either a type identifier declared in the type section of the module or a type formed by a type constructor. For example, we can declare the store account_file in Figure 4.37 as:

 student_files: **set of** Address;

where Address is a type identifier, presumably already declared in the type section, and **set of** is the type constructor for set types to be introduced in Chapter 8.

If some variables need to be declared with the same type, a shortcut is to group them together. Suppose we want to declare variables x1, x2, and x3 with the same type **int**; we can write

 x1, x2, x3: **int**;

All the variables declared in the **var** section are called *store variables*, which represent part of the state of the module. A state of a module is formed by all the store variables and available data flow variables, contained in the associated CDFD, with their values.

Note that there may be two kinds of stores in a CDFD. One is *local stores* and another is *external stores*. A local store is local to the module; it is initialized by the *InitializationProcess* specified in the same module whenever the associated CDFD is executed. An external store is *global* to the current module; its data are taken from the high level CDFDs (this point can be difficult to understand before we study the notion of process decomposition, therefore the reader can skip this part for now and return to it when he or she has studied process decomposition in the next chapter) or outside the system under construction (e.g, an existing file or database). Therefore, its data are not changed by the local *initializationProcess*. To distinguish the external and local store variables, we put the keyword **ext** before an external variable in the declaration. For example,

 var
 ext x1, x2 : **int**;

states that x1 and x2 are external store variables of type **int**. It is also useful to distinguish the external stores that are passed over from a high level CDFD and the external stores that exist independently of the system under construction, as this would facilitate the verification of specification consistency. To this end, we use the sharp mark # to decorate those store variables representing existing stores outside the system. For example,

> **var**
> **ext** x1, #x2 : **int**;

declares that both x1 and x2 are external store variables, but x1 is taken from another high level CDFD whereas x2 is an existing store outside the system. We call the external store variables like x2 *existing external variables* and the stores they denote *existing external stores*.

Another important part of a module is the invariant section *Typeand-StateInvariants*, starting with the keyword **inv**. This section may include a list of invariants, separated by semicolons, as illustrated below:

> *Invariant_1*;
> *Invariant_2;*
>   *...*
> *Invariant_v*;

Each *Invariant_i* ($i = 1..v$) is a predicate, mostly a quantified predicate; it expresses a property of types declared in the **type** section, variables declared in the **var** section, or data flow variables used in the associated CDFD. These properties must be sustained throughout the entire specification by related processes. If a property is applicable to values of a type, it should be defined as an invariant of the type. If a property is, however, only applicable to specific variables, it should be defined as an invariant of the specific variables. As an example, we define the following invariants on type Address and store variable student_files:

> **inv**
> **forall**[x: Address] | **len**(x) <= 50;
> **card**(student_files) <= 1000;

The first invariant specifies that every value of type Address must be no longer than 50 characters. The second invariant requires that the store student_files holds at most 1000 student files. The operator **len**() is defined on **string** type and sequence types: it yields the length of the string or sequence (e.g., x). The operator **card**() is defined on set types: it yields the cardinality of the set provided as the argument (e.g., student_files). The detailed introduction of

string and sequence types is given in Chapter 9 and set types are discussed in Chapter 8.

Note that the invariants specified are supposed to hold through the entire specification. This means that whenever the variable student_files, for example, is used in the related process specifications, the constraint on it, defined by the invariant **card(student_files)** $<=$ **1000**, must not be violated by the process specification. Similarly, whenever type **Address** is used to declare any variable in the module, for example,

place: Address;

the variable place is assumed to hold the property **len(place)** $<=$ **50**. Therefore, it is important to ensure that the invariant is not violated by the pre and postconditions of related process specifications given in the module. In other words, all process specifications must be kept consistent with the related invariants.

The most important part in a module is the description of its behavior. Such a description is given by a CDFD. In order to associate the CDFD with the module, we use the keyword **behav** to indicate the section where the number of CDFD is provided in the format CDFD_no, where no is the number of the CDFD. For example,

behav CDFD_10;

indicates that the associated CDFD is numbered 10 in the entire specification (an entire specification may contain many CDFDs in a CDFD hierarchy; see more details in the next chapter).

The next important part of a module is to specify all the processes occurring in the associated CDFD. Apart from the processes used in the CDFD, an additional process, called Init, also needs to be defined for initialization of the local store variables, denoted by *InitializationProcess* in the module structure outline given previously. This initialization process has the same structure as other processes, such as *process_1*, *process_2*, etc., but the function is used to initialize all the local store variables. This initialization process is a little *exceptional* because it has neither input data flows nor output data flows. For a normal process contained in a CDFD, this is disallowed, but as an initialization process of a module, it is a legal process. It is the convention not to draw the process Init in CDFDs.

As far as the process specification is concerned, we have seen many ways of specifying various processes, but what has been discussed so far does not cover all of the aspects of a process specification. The general structure of a process specification is given as follows:

process *ProcessName*(*input*) *output*
ext *ExternalVariables*

    pre *PreCondition*
    post *PostCondition*
    decom *LowerLevelModuleName*
    explicit *ExplicitSpecification*
    comment *InformalExplanation*
    end_process

Since we have been discussing the name, input, output, external variables, precodition, and postcondition of a process, we assume that the reader is already familiar with those notions. Therefore, we focus only on the explanation of the **decom**, **explicit**, and **comment** parts here. The keyword **decom** indicates that the following name is the name of the lower level module that is the decomposition of the current process *ProcessName*. The *LowerLevelModuleName* is usually composed of two parts, connected by an underscore, such as

    *ProcessName*_decom

The *ProcessName* is the same as the name of this process itself, and decom is a common suffix that is adopted by every module name in SOFL specifications, except the very top level module, as you will see in the detailed explanation about this point in the next chapter. Thus, which module is the decomposition of which process will be very clear to readers of the specification.

    Sometimes, a process may not be suitable to be specified using pre and postconditions, especially when an *object* of a *class* is involved in the input and output of the process. In this case, the process may be defined by an *explicit specification*, which is written following the keyword **explicit**. Furthermore, explicit specifications are also adopted to describe detailed design. The detailed discussion of explicit specifications are given in Chapter 6, and those of object and class are given in Chapter 13.

    To help in the understanding of a process specification, an informal explanation can be very useful. The keyword **comment** starts the informal explanation that ends before the keyword **end_process** in the process specification. Basically, one can use any character and format to express one's comment, but it is important to bear in mind that the role of such a comment is to explain what the formal specification, given in terms of pre and postconditions, means. This part can be very helpful for communication between the specification writer and the end-user of the system, since we should not expect the end-user to understand the formal notation.

    Note that except for the keywords **process** and **end_process**, as well as the process name in a process specification, all the other parts are optional, depending on the necessity in specific processes.

    The final section of a module is function definitions. As many functions as necessary can be defined in a module. Functions can be applied in predicate expressions wherever they are used, such as in invariants, preconditions,

postconditions, and the explicit specifications. As the format of function definitions is discussed in detail in Section 4.16, and functions are not involved in the discussions before that section, so let us ignore this topic for now.

4.14 How to Write Comments

A comment is useful in improving the readability of specifications. An informal comment can be given as part of a process specification to explain the functionality of the process expressed with the formal notation. In addition, there is also a need to explain the meanings of defined types, data flow variables, store variables, or whatever the entities of interest in a module. These data items are usually defined in several different places in the module; therefore, it would not be very helpful to explain them in one specific place (because the explanations far away from the data items may not be effective in helping the reader understand the data items).

SOFL has adopted another structure for comments similar to that used in the programming language C. Such a comment is written between a pair of slash-asterisk symbols /* ... */, and it can be written anywhere in a module. It does not contribute to the semantics of the module, but just provides help for understanding. For example, in the variable declaration

```
var
 student_files: set of Address;
    /*student_files is defined as a collection of home addresses, and each
      address is represented by a string. */
```

the comment explains the structure of the variable student_file and the potential meaning of the element type Address.

By now we have introduced all the sections constituting a module, although the detailed discussions of some parts are left for later chapters. However, we should bear in mind that not necessarily all the sections are defined in a module, though it is possible. Some modules may have constant and type declarations, but with no invariants, while some other modules may have type declarations, but with no variable declarations and invariants. In fact, except for the module name and process specifications, all the other parts are optional, depending on the need of specific modules.

4.15 A Module for the ATM

We take the simplified ATM modeled as the CDFD in Figure 4.37 as an example to show a complete picture of a module. The way to build a module usually starts from the construction of the CDFD, because it expresses the potential behavior and the architecture of the module. Then all the data

stores, processes, and data flows used in the CDFD are defined. The module for the ATM is given as follows:

module SYSTEM_ATM /* This module has no parent module.*/
type
Account = **composed of**
        account_no: **nat**
        password: **nat**
        balance: **real**
        end

  var

  ext #account_file: **set of** Account;
   /* the account_file is an external store that
      exists independently of the ATM system.
    It is defined as a set of accounts. */
  inv

  forall[x: Account] | 1000 <= x.password <= 9999;
   /* The password of every account must be a
      natural number with four digits. */

  behav CDFD_1;
  /*Assume the ATM CDFD in Figure 4.37 is numbered 1.*/

  process Init()
  end_process;
  /* The initialization process does nothing because there is no
    local store in the CDFD to initialize. In this case, this process
    can be omitted.*/

  process Receive_Command(balance: **sign** | w_draw: **sign**) sel: **bool**
  post balance <> **nil and** sel = **true or** w_draw <> **nil and** sel = **false**
  comment
  This process recognizes the input command show balance or withdraw
  cash. The output data flow sel is set to true if the command is showing
  balance; otherwise, if the command is withdrawing cash, sel is set to false.
  end_process;

  process Check_Password(card_id: **nat**, sel: **bool**, pass: **nat**)
                account1: Account | pr_meg: **string** |
                account2: Account
  ext rd account_file /*The type of this variable is omitted because

this external variable has been declared in
the **var** section. */

post (**exists!**[x: account_file] |
  x.account_no = card_id **and**
  x.password = pass **and**
  (sel = **false and** account1 = x **or**
  sel = **true and** account2 = x)

  or

  not (**exists!**[x: account_file] | x.account_no = card_id **and**
  x.password = pass) **and**
  pr_meg = "Reenter your password or insert the correct card"

comment

If the input card_id and pass are correct with respect to the existing information in account_file, then if sel is false, the account information is passed to the output account1; otherwise, the account information is passed to the ouput account2. However, if one of card_id and pass is incorrect, a prompt message pr_meg is produced.

  end_process;

  process Withdraw(amount: **real**, account1: Account)
          e_msg: **string** | cash: **real**
  ext wr account_file
  pre account1 **inset** account_file
        /*input account1 must exist in the account_file*/
  post (**exists**[x: account_file] | x = account1 **and**
  x.balance >= amount **and**
  cash = amount)

    and
  account_file = **union**(**diff**(~account_file, {account1}),
    {**modify**(account1, balance -> account1.balance - amount)})
   or
  not exists[x: account_file] | x = account1 **and**
  x.balance >= amount **and**
  e_meg = "The amount is too big")

  comment

The required precondition is that input account1 must belong to account_file. If the request amount to withdraw is smaller than the balance of the account, the cash will be withdrawn. On the other hand, if the request amount is bigger than the balance of the account, an error message "The amount is too big" will be issued.

  end_process;

  process Show_Balance(account2: Account) balance: **real**
  post balance = account2.balance;
  end_process;
  end_module;

Since there is no local store in the CDFD of this module, the initialization process Init does not do anything. Some relatively complicated process specifications are explained informally in the comment parts, such as for the processes Receive_Command, Check_Password, and Withdraw, but for a simple process like Show_Balance no comment is provided. There is no need to specify the external process Customer, because its role is just to help understand the CDFD.

Note that several operators defined in set types and composite types are used, such as **union**(), **diff**(), **modify**(), etc. Briefly speaking, the operation **union**(x, y) is the union of the two sets x and y; **diff**(x, y) yields the set whose elements belong to x but not y; and **modify**(x, f -> v1) yields a new composite object from the given composite object x by replacing the value of its field f with v1. The detailed discussion of these operators will be given in Chapters 8 and 10, respectively.

4.16 Compound Expressions

Writing a process specification with the predicate expressions introduced so far may result in a poor readability of the specification due to their complicated structures. For example, the postcondition of process Check_Password in module SYSTEM_ATM in the previous section actually defines three exclusive conditions: the requested account exists and sel = **true**; the account exists and sel = **false**; and the account does not exist, but the description does not seem to be obvious because of the complicated structure of the expression. To achieve good readability of specifications, more explicit syntax may provide effective help.

In this section, we introduce several compound expressions for this purpose. These expressions include if-then-else, let, and case expressions.

4.16.1 The if-then-else Expression

An **if-then-else** expression is a conditional expression that yields a result based on the value of the guard condition involved in the expression. The general format is

if B **then** E_1 **else** E_2 .

If the guard condition B is true, the result of this conditional expression is E_1; otherwise, if B is false, the result is E_2. Let result denote the conditional expression; then the expression is equivalent to the predicate:

B **and** result = E_1 **or not** B **and** result = E_2

For example,

if x > 5 **then** x + z **else** z - x

or, the one with better readability,

if x > 5
then x + z
else z - x

4.16.2 The let Expression

The **let** expression is used to declare some identifiers denoting expressions in predicate expressions. Two **let** expressions are designed for this purpose. The first **let** expression takes the format

let $v\_1 = E\_1, v\_2 = E\_2, ..., v\_n = E\_n$ **in** $P(v\_1, v\_2, ..., v\_n)$

where n >=1.

In this expression, each v_i (i=1..n) is an identifier that serves as a *pattern* other than a variable (whose value may change), because it has no function for holding any value; it just denotes the corresponding expression E_i. P is a predicate expression in which patterns v_i are involved. Semantically, this **let** expression is equivalent to the following expression:

$$P[E\_1/v\_1, E\_2/v\_2, ..., E\_n/v\_n]$$

This substituted expression is derived from substituting E_i for v_i (i=1..n) in expression P. Note that brackets ( ) are used to enclose E_i (i=1..n) if any ambiguity in interpreting the substituted predicate expression occurs.

Consider the following **let** expression:

let x1 = y + z * * 2, x2 = y - z * 5 **in**
a* x1 ** 2 + b * x1 + c > a * x2 ** 2 + b * x2 + c

It is equivalent to the substituted expression:

a * (y + z * * 2) ** 2 + b * (y + z * * 2) + c >
a * (y - z * 5) ** 2 + b * (y - z * 5) + c

Another **let** expression has the following format:

let x: T | R(x) **in** P(x)

This **let** expression introduces a pattern x that is bound to a value of set T (which may also be a type) satisfying condition R(x). Pattern x is usually involved in P. For example,

> **let** x: nat | x > 5 **in** y > x + 1

To keep this kind of **let** expression simple, we do not allow the introduction of multiple binding, such as x1: T1, x2: T2. If such a multiple binding is really needed in an expression, we can use the **let** expression several times, as

> **let** x_1: T1 | R1(x_1) **in**
> **let** x_2: T2 | R2(x_2) **in** P(x_1, x_2)

Also, condition R(x) may be omitted in a **let** expression so that pattern x will be introduced as any value in type T with no constraint. Thus, the following format can also be used:

> **let** x: T **in** P(x)

For example,

> **let** x: **nat in** y > x + 1

describes a different predicate.

4.16.3 The case Expression

A **case** expression is a multiple conditional expression. Its format is as follows:

> **case** x **of**
> ValueList_1 –> E_1;
> ValueList_2 –> E_2;
>   ...
> ValueList_n –> E_n;
> **default** –> E_n + 1
> **end_case**

where each ValueList_i (i=1..n) is a list of concrete values of the same type as that of x. x can be a variable, a pattern defined in a **let** expression, or an expression of any type.

The **case** expression states that if x is equal to one of the *values* in ValueList_1, the result of this expression will be E_1; otherwise, if x is one of the values in ValueList_2, the result of this expression will be E_2; and so on. However, if x is different from all of the values given in ValueList_1, ValueList_2, ..., ValueList_n, the result of this expression will be E_n + 1 as

default. Note that all of the values in ValueList_1, ValueList_2, ..., ValueList_n should be disjoint; but even if they are not, the case expression will not involve ambiguity in obtaining the final result, because ValueList_1, ValueList_2, ..., ValueList_n are evaluated in order, and once one of them matches x, the corresponding expression E_i (i=1..n) will be taken as the final result of the case expression. The **default** clause may not be used if ValueList_1, ValueList_2, ..., ValueList_n cover all of the possible cases. However, having a **default** clause is always recommended for it will ensure a defined **case** expression. For example,

```
case x of
1, 2, 3 -> y + 1;
4, 5, 6 -> y + 2;
7, 8, 9 -> y + 3;
default -> y + 10
end_case
```

4.16.4 Reference to Pre and Postconditions

The pre and postconditions of an already defined process, say A, can be referenced as a predicate expression by another process, say B. Such a reference may arise when there is a need to write a predicate in the specification of process B that is exactly the same as the pre or postcondition of process A. Thus, the specification of B can be made more concise than the one without such references. However, we must bear in mind that such a reference is purely syntactical; it does not mean an invocation or execution of process A, but just the reuse of its pre or postcondition.

The symbols **pre_A** and **post_A** denote the pre and postconditions of process A, respectively. Thus, the specification of process B involving the references to the pre and postconditions of A is given as follows:

```
process B()
pre P and pre_A
post Q or post_A
end_process
```

where P and Q are two predicate expressions.

If one wants to define another process, say C, with the same input and output data flows, external variables, and the functionality as those of process A, one can define C as follows:

```
process C equal A
end_process
```

The only difference between A and C is that they have different names, and all of the other parts have the same syntax. However, it is worth noting that

this does not mean that executing process C is equivalent to executing process A. It really means that process C has the same syntax as process A, except for its name. If A is defined in another module, say M, then we need to write M.A to refer to A. Thus, process C above can be defined as follows:

> **process** C **equal** M.A
> **end_process**

It is sometimes necessary to directly use process A, which is defined in module M, in the current CDFD. In this case, process A still needs to be defined in the associated module of the current CDFD for the consistency between CDFD and its module and for the readability of the entire specification. The form of the specification of process A occurring in the current CDFD is

> **process** A **equal** M.A
> **end_process**

4.17 Function Definitions

A function provides a mapping from its domain to its range. A function differs from a process in several ways:

- A function does not allow exclusive inputs and outputs, whereas a process does.
- A function yields only one output, whereas a process allows many outputs.
- A function does not access external variables (like stores in CDFDs), whereas a process may do so.

4.17.1 Explicit and Implicit Specifications

There are two ways to define a function: *explicit specification* and *implicit specification*. An explicit specification shows how a function can actually be computed, whereas an implicit specification defines the function using pre and postconditions.

The format of an explicit specification is

> **function** $Name(InputDeclaration) : Type$
> $== E$
> **end_function**

where $Name$ denotes the function name; $InputDeclaration$ is the parameter declaration (which can be empty); $Type$ denotes the range of the function; and E is an expression of any type available in SOFL (e.g., **int**, **real**, **bool**),

and is called the *body* of the function. For example, function **add** is defined explicitly as follows:

> **function add(x, y: int) : int**
> == x + y
> **end_function**

The format of implicit specification of a function is:

> **function** *Name(InputDeclaration)* : *Type*
> **pre** *Pre*
> **post** *Post(Name)*
> **end_function**

where *Post(Name)* is a predicate expression that must involve *Name*. In fact, we use *Name* as the output variable to represent the result of the function. For example, the **add** function is defined implicitly as follows:

> **function add(x, y: int) : int**
> **pre true**
> **post** add = x + y
> **end_function**

where **add** is used as a variable to hold the result of the function.

As the reader will see in later chapters, implicit specifications are usually constructed for abstract designs, and then refined into explicit specifications during detailed designs. In order to allow the developer to maintain the history of specification evolution over different development phases, the mixture of the implicit and explicit specifications in a function definition is allowed. That is, it is possible for a function to have both pre and postcondition and the explicit expression. Furthermore, since it is possible for a function to remain undefined completely in the specification for some reason, a function with only a signature is also allowed. In this case, how the function is defined will depend on the developer's decision made at the subsequent development phase. A way to claim that a function is undefined in the specification is to use the keyword **undefined**, like function A:

> **function A(x, y: int) : int**
> == **undefined**
> **end_function**

4.17.2 Recursive Functions

A recursive function is a function that applies itself during the computation of its body. When writing a specification for a recursive function, two points are important:

- the body of the function (for explicit specification) or the postcondition of the function (for implicit specification) must contain an application of the function.
- an *exit* is necessary to ensure that any application of the function terminates.

Let us take the factorial function fact as an example. This function takes a natural number n and computes its factorial n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1. The explicit specification is:

function fact(n: **nat**) : **nat**
== **if** n = 1
    then n
    else n * fact(n - 1)
end_function

In the body of this function fact(n - 1) is an application of the same function, but to a different argument n - 1. The condition n = 1 provides an exit for this function.

The implicit specification is:

function fact(n: **nat**) : **nat**
post if n = 1
    then fact = n
    else fact = n * fact(n - 1)
end_function

The precondition is true (and therefore omitted). In the postcondition fact = n is the exit provided by the condition n = 1. The application of fact itself is involved in the expression fact = n * fact(n - 1), where fact denotes the result of the function, while fact(n - 1) is an application of the function to argument n - 1.

4.18 Exercises

1. Answer the following questions:
   a) What is a process?
   b) What is a data flow?
   c) What is the difference between active data flows and control data flows?
   d) What is a data store?
   e) What is the difference between data stores and data flows?
   f) What are the conditional structures for?
   g) What are the merging and separating structures for?
   h) What are the diverging structures for?
   i) What are the connecting structures for?
   j) What is a condition data flow diagram (CDFD)?
   k) What is a module for?
   l) What is the general structure of a module?
   m) What is an invariant?
   n) What is the general structure of a process?
   o) How to make a reference to the precondition or postcondition of a process?
   p) What is a function?
   q) What is the difference between a process and a function?
   r) What are the general formats of explicit and implicit specifications of a function?
   s) What is a recursive function, and what are the important points in writing recursive functions?

2. Define a calculator as a module. Assume that **reg** denotes the register that is accessed by various operations. The operations include **Add**, **Subtract**, **Multiply**, and **Divide**. Each operation is modelled by a process.

3. Write a module defining all the data flows, stores, and processes of the CDFD in Figure 4.38, assuming all the data flows and stores are integers, and all the processes perform arithmetic operations.

4. Change the following compound expressions into equivalent predicate expressions.
   a) **let** a = x + y, b = z + w **in** a ****2 *** b + b ***** y ***** w
   b) **if** x > 0 **then** a = x + 1 **else** a = x + 10
   c) a = **case** x of 1, 2, 3 –> x +1; 4, 5 –> x + 2; 6 –> x ***** x; **default** –> x **end**

5. Write both the explicit and implicit specifications for the function Fibonacci:

   Fibonacci(0) = 0;
   Fibonacci(1) = 1;
   Fibonacci(n) = Fibonacci(n - 1) + Fibonacci(n - 2)
   Where n is a natural number of type **nat0**.
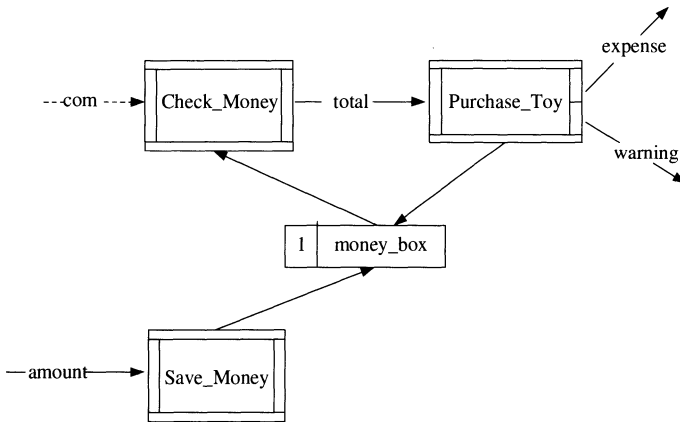
com: command for checking the total amount of the money in the money-box

amount: the amount of money to be saved in the money_box

total: the total amount of the money in the money_box

expense: the sufficient amount for purchasing a toy

warning: a warning message for the shortage of the money in the money_box

Fig. 4.38. The CDFD for problem 3 in Exercise 4

5

Hierarchical CDFDs and Modules

When building a specification for a complex system, it is almost impossible to construct only a one level CDFD and module. A complex system needs many processes, data flows, and data stores, as well as other structures. If we try to draw them in one CDFD, the diagram would be too large to manage. Also, the efficiency of constructing, editing, and reading such a complex CDFD would be very bad. Furthermore, building a large and complex software system usually involves many developers, and they need to share the job by taking care of different parts of the entire system. How to organize a large CDFD so that all of the developers can cooperate efficiently and responsibly becomes a very important issue to address. Even if a system is built by a single developer, the problem of how to reduce the complexity of a one level CDFD and the entire specification is still a major concern. The solution of SOFL to this problem is to support the construction of hierarchical CDFDs and the associated modules by *process decomposition*.

In this chapter, the reader is expected to study how a process is decomposed into a CDFD; what the relation is between a process and its decomposition; and how hierarchical CDFDs and their associated modules are organized in a consistent manner, so that we can form supportive connections between logically related components in the entire CDFD and module hierarchies.

5.1 Process Decomposition

Process decomposition is an activity to break up a process into a lower level CDFD. The CDFD defines in detail how the input data flows of the process are transformed into its output data flows through other intermediate processes or structures. Let us take the process Check_Password in Figure 4.37 in Chapter 4 as an example to explain what process decomposition exactly means. To facilitate the discussion, and to help us concentrate on this particular process rather than the entire CDFD given in Figure 4.37, process Check_Password and the associated store account_file are redrawn in Figure 5.1.
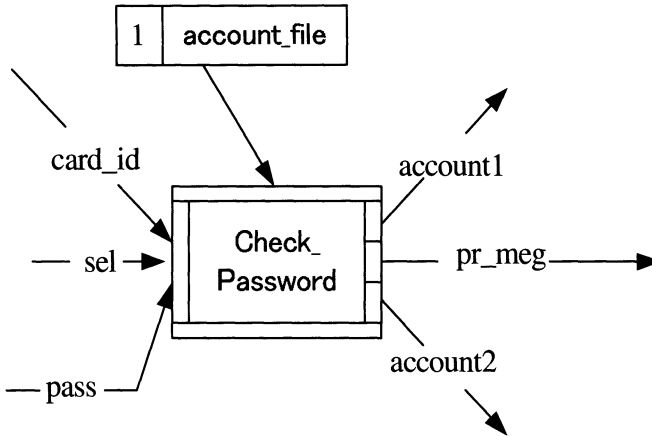
Fig. 5.1. Process Check_Password

As defined by its specification in module SYSTEM_ATM given in Chapter 4, this process receives card_id, sel, and pass, and checks whether or not the requested account exists in store account_file. If it does, and the value of sel is **true**, the account information will be supplied as the output data flow account1. If this account exists but the value of sel is **false**, the account information will be provided as the output data flow account2. However, if this account is not included in account_file, the prompt message is given to suggest to the customer what to do next. A refinement of this process specification is to decompose the process into a lower level CDFD that gives more algorithmic information about how the functionality of this process is realized via other intermediate processes. The decomposed CDFD of process Check_Password is given in Figure 5.2.

In this CDFD, the input data flows card_id and pass are checked by process Confirm_Account against store account_file to confirm validity of the requested account. If this account is a valid one, its information is transferred to data flow account; otherwise, the prompt message pr_meg is generated. If account is available, then it will be transferred to data flow account1 or account2 by process Transfer_Account, depending on the truth value of input data flow sel.

Definition 12. *If a process A is decomposed into a CDFD, we call the CDFD the decomposition of process A, and process A, the high level process of the CDFD.*

Thus, the CDFD in Figure 5.2 is the decomposition of its high level process Check_Password in Figure 5.1. Such a decomposed CDFD must also be associated with a module, in a way that this CDFD represents the behavior of the module, and all data flows, stores, and processes occurring in this CDFD are
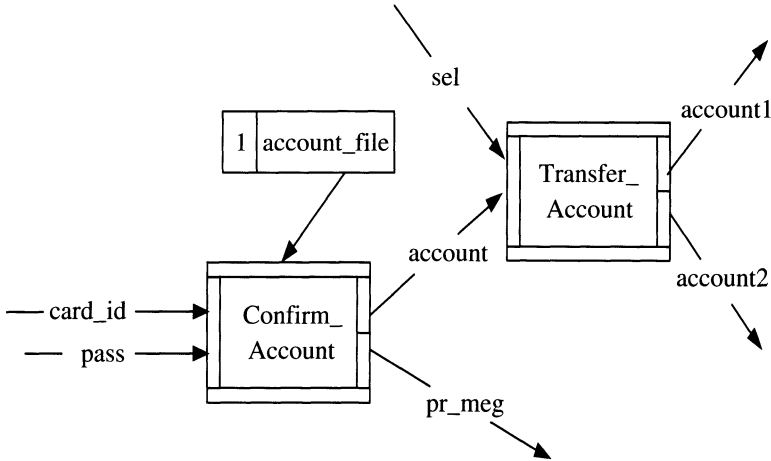
Fig. 5.2. The decomposition of process Check_Password

specified in the module. The outline of the module associated with the CDFD in Figure 5.2 is given as follows:

> **module** Check_Password_decom / SYSTEM_ATM;
> **type**
> Account = SYSTEM_ATM.Account;
> **var**
> **ext** account_file: set of SYSTEM_ATM.Address;
> **behav** CDFD_2; /* Assume the CDFD in Figure 5.2 is numbered 2. */
> **process** Confirm_Account(card_id: **nat0**, pass: **nat0**)
>   account: Account | pr_meg: **string**
> ...
> **end_process**;
> **process** Transfer_Account(sel: **bool**, account: Account)
>   account1: Account | account2: Account
> ...
> **end_process**;
> **end_module**;

This module is named Check_Password_decom, and its parent module in which process Check_Password is defined is SYSTEM_ATM. The type Account is declared in terms of the already declared type Account in module SYSTEM_ATM. We use SYSTEM_ATM.Account to refer to the type identifier Account declared in module SYSTEM_ATM, and likewise for SYSTEM_ATM.Address. As far as the reference of type identifiers and other components of a module is concerned, we will elaborate it in Section 5.5. Note

that the declared type Account in module Check_Password_decom is referenced whenever it is used within this module. For example, the output data flow account of process Confirm_Account is declared as a variable of type Account, and this Account refers to the Account declared in the type section of this module, not the Account declared in module SYSTEM_ATM; although they are declared as the same type in this case.

The advantage of the declaration of type Account in this module is that it makes the data flow variable or store variable declarations concise. For example, suppose we want to declare three data flow variables account (output data flow of process Confirm_Account), account1, and account2 (output data flows of process Transfer_Account) with the type Account defined in module SYSTEM_ATM, if the direct reference SYSTEM_ATM.Account is adopted, the process specifications will become much longer, and perhaps tedious. Of course, if a type declared in another module is only used occasionally, the direct reference of the type may be appropriate to use, like SYSTEM_ATM.Address, which is used only once to declare the external store variable account_file.

The outlines of the specifications of processes Confirm_Account and Transfer_Account do not involve any new feature, so the reader is supposed to have no problem in understanding them and, therefore, the entire module.

In principle any process in the CDFD of Figure 5.2 may be decomposed again if necessary. If such decompositions continue, the hierarchies of CDFDs and the associated modules will be created. Figure 5.3 depicts a hierarchy of three level CDFDs. The top level CDFD, numbered 1, is composed of four processes A1, A2, A3, A4, and a data store s1. For some reason, process A1 is decomposed into the CDFD 2, while process A3 is decomposed into the CDFD 3. Finally, process A33 in CDFD 3 is decomposed into the lowest level CDFD 4. For each level of the CDFD hierarchy, it is necessary to provide a module to define the components of the associated CDFD. As the relation between high level processes and their decompositions are recorded in the process specifications given in the modules, a hierarchy of the associated modules is actually also created. Both the CDFD hierarchy and the module hierarchy constitute the entire specification.

The associated module hierarchy of this CDFD hierarchy is outlined as follows:

module SYSTEM_Example;
...
var
s1: Type1;

behav CDFD_1;
process Init;
process A1
decom: A1_decom;
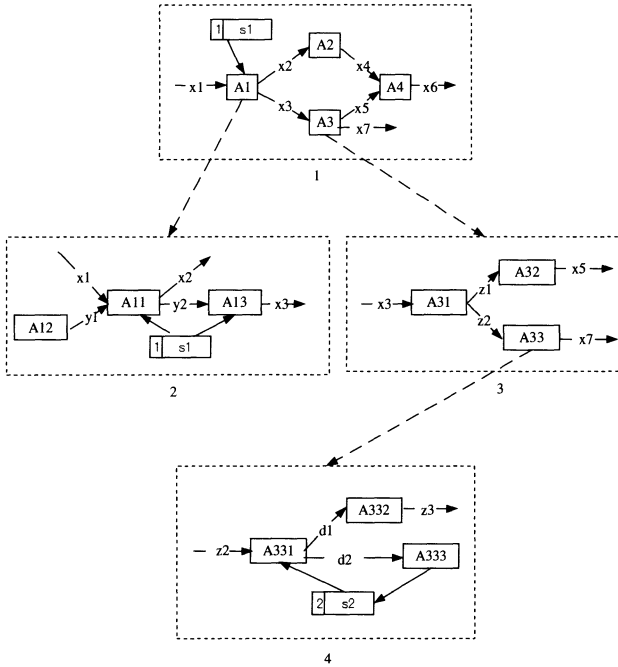
Fig. 5.3. The outline of a hierarchy of three level CDFDs

```
/*Module A1_decom is associated with the decomposition
  of A1. */
end_process;
process A2;
process A3
decom: A3_decom;
/*Module A3_decom is associated with the decomposition
  of A3. */
end_process;
process A4;
end_module;


module A1_decom;
...
var
ext s1: Type1;

behav CDFD_2;
process Init;
process A11;
```

process A12;
process A13;
end_module;

module A3_decom;
...

behav CDFD_3;
process Init;
process A31;
process A32;
process A33
decom: A33_decom;
/*Module A33_decom is associated with the
  decomposition of A33. */
end_process;
end_module;

module A33_decom;
...
var
s2: Type2;

behav CDFD_4;
process Init;
process A331;
process A332;
process A333;
end_module.

The four modules are separated by semicolons, but the last module ends with a period, indicating the end of the module definitions in the entire specification. In each module are only those processes that are decomposed into CDFDs provided with more details about the modules associated with their decompositions, while the rest of the processes are represented simply by their names. For example, process A1 is indicated as having a decomposition associated with module A1_decom, while process A3 has a decomposition associated with module A3_decom.

From the viewpoint of operational semantics, a high level process is equivalent to its decomposition. In other words, the execution of a high level process is actually performed by executing its decomposition. Also, from the functional semantics point of view, the high level process is defined by its decomposition in the manner that the decomposition satisfies the specification of the process. This property is known as *correctness* of the decomposition against its high level process specification.

There are many issues concerned with building the hierarchies of CDFDs and modules. For example, how stores, and input and output data flows of a high level process should be handled in the decomposition; how the notion correctness should be defined so that it can provide a guideline for decomposing processes; how the scope of declared data items, such as constants, types, and store variables, in each module should be defined; and how process specifications and/or invariants defined in another module should be reused. We discuss these issues one by one from the next section.

5.2 Handling Stores in Decomposition

When a process is decomposed into a CDFD, the stores connected to the process must also be consistently accessed by some lower level processes occurring in the decomposition. This problem has two aspects. One is that the store accessed by the high level process must sink into its decomposition, and another is that the store should be accessed consistently with the way it is accessed by the high level process.

For example, process A1 in CDFD 1 of Figure 5.3 reads from the store s1. When A1 is decomposed into CDFD 2, store s1 is passed over to this lower level CDFD and also read by some lower level processes, like A11 and A13.

Generally speaking, a store accessed by a high level process must also be drawn in the decomposition of the process, and must be accessed in the same way, probably by several processes. On the other hand, all the *external* stores (*except the existing external stores*) occurring in the decomposition of a high level process must occur in the high level CDFD (the CDFD in which the high level process is used). That is, all the external stores must be sunk from the high level CDFD, and all the other stores, if any, must be *locally defined* stores. Since this property constitutes part of the structural consistency of hierarchical CDFDs, it is important to formalize this notion by giving an appropriate rule.

Let G denote a CDFD. Then we define the following notation that will be used in the formalization.

Notation:

Store(G) = the set of all the stores occurring in CDFD G.
Store_ext(G) = the set of all the external stores occurring in CDFD G.
Store_loc(G) = the set of all the local stores occurring in CDFD G.
Store_acc(A) = the set of all the stores which process A accesses.
Acc_p(A, s) **inset** {**rd**, **wr**}, denoting the way of accessing store s by process A.
Acc_d(G, s) **inset** {**rd**, **wr**}, denoting the way of accessing store s by CDFD G.

Obviously, the union of Store_ext(G) and Store_loc(G) must be the same as Store(G). Furthermore, Acc_d(G, s) = **rd** means that store s is possibly read by some process in G, but definitely not updated by any processes in G; and Acc_d(G, s) = **wr** if there is a possibility that s is updated by some process in G, which does not eliminate the possibility of reading from s by some processes in G.

Rule 5.1 Let A be a high level process in CDFD G_h and Store_acc(A) = {s_1, s_2, ..., s_n}. Let CDFD G_d be the decomposition of process A. Then, the following conditions must hold:

(1) Store_acc(A) is a subset of Store_ext(G_d)
(2) **forall**[s: Store_ext(G_d)] | Acc_d(G, s) = Acc_p(A, s).

Note that we treat all the external stores representing external devices or files, such as displays, files on disks, printers, and keyboards, as existing external stores (decorated with the sharp mark # when they are declared in a module), since they usually exist independently of the software system under construction. All the existing external stores are considered as global variables to all the CDFDs in a CDFD hierarchy. In principle, such an existing external store must occur in every related CDFD (in which it is accessed), but sometimes you may not want to show its access by a high level process in the high level CDFD, while there may be a need to show its access by some processes in the decomposition of the high level process. In that case, one may draw the existing external store in the decomposition without drawing it in the high level CDFD. This principle is reflected by condition (1) in Rule 5.1.

For example, the store s2 of CDFD 4 in Figure 5.3 is treated as an existing external store; it does not occur in CDFD 3, in which the high level process A33 of CDFD 4 is included.

This way of dealing with existing external stores can benefit the construction of CDFD hierarchies in two ways. One way is by allowing the specification writer to concentrate on the most important and necessary issues related to the high level process and other processes in the high level CDFD, and to put the existing external stores where they are most properly used. Another way is by helping avoid the unnecessary drawing of stores.

It is worth noting that Rule 5.1 does not restrict the use of local stores in the decomposition of a high level process. This implies that as many local stores as necessary can be declared and used in a CDFD.

5.3 Input and Output Data Flows

Since a high level process is actually represented by its decomposition, the input data flows, and the output data flows of the process and its decomposition must be kept consistent. If all input and output data flows of the high level process are *the same as* input and output data flows of the decomposition,

the high level process and its decomposition are said to be consistent in their input and output data flows.

To define formally this consistency property, we need the following notation:

Notation:

Port_i(A) = the set of input ports of process A.
Port_o(A) = the set of output ports of process A.
Dataflow_i(A, P) = the set of input data flows connected to the input
                    port P of process A.
Dataflow_o(A, P) = the set of output data flows connected to the output
                    port P of process A.
Input_p(A) = the set of the input data flows of process A.
Output_p(A) = the set of the output data flows of process A.
Input_d(G) = the set of all the input data flows of CDFD G.
Output_d(G) = the set of all the output data flows of CDFD G.

Obviously, Input_p(A) is the union of all the data flows connected to all the input ports of A, and Output_p(A) is the union of all the data flows connected to all the output ports of A.

Definition 13. *Let G be a CDFD; A_1, A_2, ..., A_n be all its starting nodes; and B_1, B_2 , ..., B_m be all its terminating nodes. Then, Input_d(G) = union (Input_p(A_1), Input_p(A_2), ..., Input_p(A_n)) and Output_d(G) = union (Output_p(B_1), Output_p(B_2), ..., Output_p(B_m)).*

where all the starting and terminating nodes include starting and terminating processes.

This definition states that the input data flows of CDFD G are the same as those of all the starting nodes of G, and the output data flows are the same as those of all the terminating nodes of G.

Rule 5.2 Let process A be decomposed into CDFD G. Then, conditions (1) and (2) given below must be satisfied.
   (1) Input_d(G) = Input_p(A)
   (2) Output_d(G) = Output_p(A)

If conditions (1) and (2) are satisfied by process A and its decomposition G, we say A and G are *consistent in their input and output data flows*.

Note that this rule only suggests very simple checking on the consistency of input and output data flows of a high level process and its decomposition. Since the high level process may have several input or output ports, and the data flows connected to different ports cannot be used together in executing

High level process

The decomposition of process W
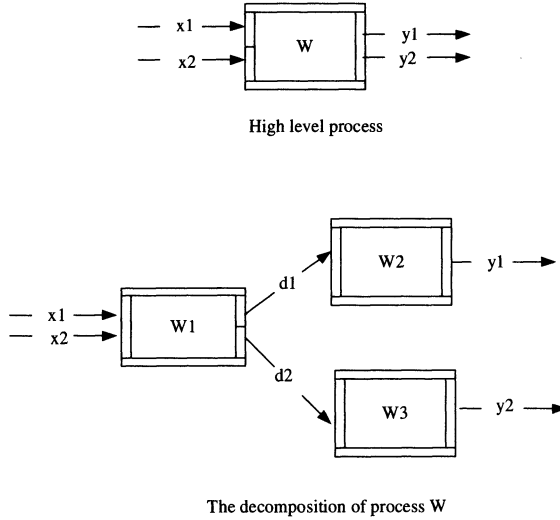
Fig. 5.4. An inconsistent decomposition of process W

the process and its decomposition, the rule does not ensure that the high level process is implemented correctly by its decomposition.

For example, Figure 5.4 shows a decomposition of process W. To execute W, input data flows x1 and x2 must be used exclusively, but in the execution of its decomposition both x1 and x2 are required by process W1. Furthermore, in process W, output data flows y1 and y2 are both generated as the result of its execution, but only one of y1 and y2 can be generated by the decomposition of process W, due to the exclusive generation of the intermediate data flows d1 and d2.

In fact, such an inconsistency is a semantic problem rather than a syntactical problem, and checking this kind of problem can be done by verifying the correctness of the decomposition of process W. Before discussing the notion correctness, we need to define the *structural consistency* first.

Definition 14. *Let process A be decomposed into CDFD G. If A and G satisfy both Rule 5.1 and 5.2, we say that A and G are structurally consistent.*

Definition 15. *Let H_g be a hierarchy of CDFDs. If every high level process and its decomposition are structurally consistent, we say that the hierarchy of CDFD H_g is structurally consistent.*

The structural consistency is a necessary condition for the decomposition of a high level process to perform what the high level process requires in its specification, but not a sufficient condition, as we have analyzed previously.

5.4 The Correctness of Decomposition

The decomposition of a high level process can be perceived as an implementation of the process, in the sense that the decomposition is intended to provide the functionality of the high level process in a more "executable" style. We say that the decomposition is *correct* with respect to its high level process specification if it does exactly what is required by the high level process. To allow the verification of such a correctness, we first need to formalize the concept of correctness. To this end, several other concepts concerned with CDFD are needed.

Definition 16. *Let G be a CDFD. Then, a data flow path of G is a sequence of data flow groups traversed by an execution of G from a starting process to all the necessary terminating processes of G.*

A data flow path of G is not a static concept but a dynamic concept. Thus a data flow path is always associated with an execution of the CDFD. For example, in Figure 5.5 possible paths of the decomposition of process W are:

(1) x1; {d1, d2}; {y1, y2}
(2) x2; {d1, d2}; {y1, y2}

where data flow groups in each data flow path are separated by semicolons. All the data flows in each group (denoted as a set), like {d1, d2}, can be executed in parallel. Also, each data flow in a data flow group may also be another data flow group or sequence of data flow groups. This would be clearer if one draws a CDFD with a data flow loop.

To make the discussion of the correctness of CDFDs simple to understand, we restrict the high level process to one as simple as process W in Figure 5.5. If one is interested in further investigation, one can extend the formalization given below to a general process.

Let A be a high level process:

process A (x1: Ti_1 | x2: Ti_2) y1: To_1, y2: To_2
ext wr s: Ts
pre pre_A
post post_A
end_process
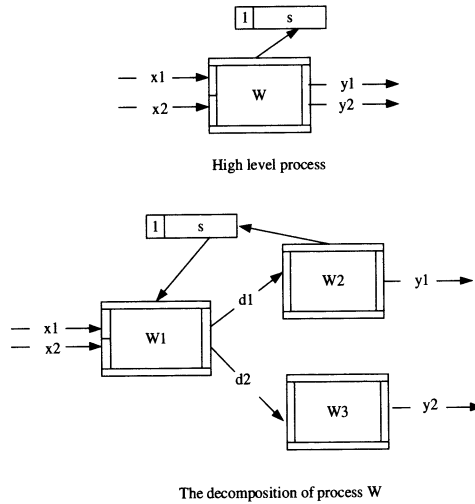
Fig. 5.5. An example of process deomposition

Let G denote the decomposition of A. Then, the correctness of G with respect to A is defined as:

Definition 17. *If A and G are structurally consistent and the following condition holds, we say that G satisfies A, or G is correct with respect to A.*

$$(\textbf{forall}[\text{x1: Ti\_1, \~s: Ts}] \mid \textbf{pre\_A}(\text{x1, x2, \~s}) =>$$
$$\textbf{post\_A}(\text{x1, x2, G(x1), \~s, s}))$$

or

$$(\textbf{forall}[\text{x2: Ti\_2, \~s: Ts}] \mid \textbf{pre\_A}(\text{x1, x2, \~s}) =>$$
$$\textbf{post\_A}(\text{x1, x2, G(x2), \~s, s}))$$

We use G(x1) (or G(x2)) to denote a set of the arbitrary output data flows generated by G (e.g., {y1, y2}), through an execution taking x1 (or x2) as its input. To obtain G(x1) (or G(x2)), we need to find a data flow path in G that starts with x1 (or x2) and ends with {y1, y2}. Note that process A and its decomposition G are structurally consistent; that is, Rules 5.1 and 5.2 are both satisfied, which is part of the condition for correctness; without them, only the quantified predicate expression may not be strong enough to ensure the real semantic consistency between the high level process and its decomposition. For example, if the pre and postconditions of process A are both **true**, and rule 5.1 is not satisfied, then a decomposition of A that does not access any store connected to the high level process A may still satisfy the

quantified predicate expression, and therefore may still be treated as a correct implementation, although it should not be.

How to verify the correctness is an interesting issue. In principle, the correctness of a CDFD can only be performed by a formal proof. However, since proofs are usually difficult, even with the use of powerful theorem provers, we suggest the use of the compromised, but practical and reasonably powerful, rigorous reviews and testing techniques. These two techniques are introduced in detail in Chapters 17 and 18.

Applying this definition to the decomposition of process W given in Figure 5.4, we can easily recognize that when x1 or x2 is available to process W, it is impossible to find a path starting from x1 or x2 that ends up with output data flows y1 and y2, because the execution of process W1 requires both x1 and x2, while process W only requires one of x1 and x2.

5.5 Scope

When defining stores or input and output variables, or writing the pre and postconditions of a process specification in a module, we may need to use types, constants, or other components, such as functions, preconditions, and postconditions of processes, declared in another module in a module hierarchy (which corresponds to a CDFD hierarchy). How to make references to those components becomes a very important issue in writing specifications. SOFL has a simple rule for this. Any constant or type identifier used in a module refers to the corresponding declaration in the same module if the declaration exists. However, if such a declaration does not exist in the current module, then it will refer to a possible declaration in its parent module; if no corresponding declaration is given in its parent module, then it will refer to a possible declaration in its grandparent module, and so on. However, if a constant or type identifier declared in a module that has no direct or indirect decomposition relation with the current module in which the constant or type identifier is used, the module containing the declaration must be indicated in the reference.

To formally define the scope of the effectiveness of declarations, we need the following notions.

Definition 18. *Let process A be defined in the module M1. If A is decomposed into a CDFD associated with module M2, we say that process A is decomposed into module M2.*

Definition 19. *If process A defined in the module M1 is decomposed into module M2, M2 is called child module of M1, while M1 is called parent module of M2.*

M

M 1   M 2   M 3

M 1 1   M 1 2   M 1 3

M 1 2 1   M 1 2 2   M 1 2 3
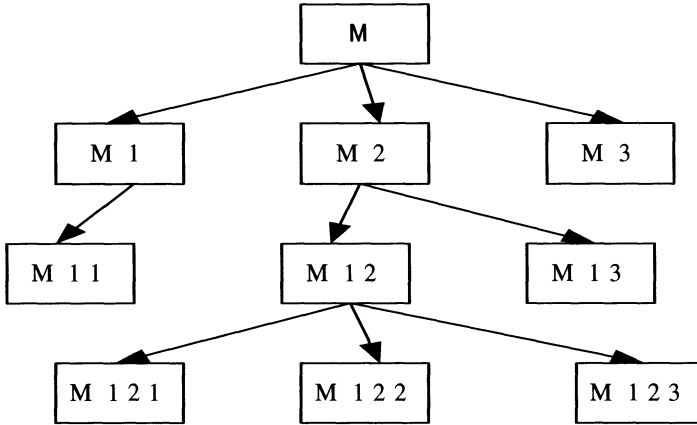
Fig. 5.6. A hierarchy of modules

Definition 20. *Let A_1, A_2, ..., A_n be a sequence of modules where n > 1. If A_1 is a parent module of A_2, and A_2 is a parent module of A_3, ..., and A_n-1 is a parent module of A_n, we call A_1 ancestor module of A_n and A_n descendant module of A_1.*

Definition 21. *If the module A is neither ancestor module nor descendant module of module B, A and B are called relative modules.*

For example, suppose that the tree given in Figure 5.6 represents a module hierarchy in which each node denotes a module. In this hierarchy, module M is an ancestor module of every other module in the tree; M2 is the parent module of M12, and an ancestor module of M121, M122, and M123. On the other hand, M121 is an descendant module of M12, M2, and M. Since M3 and M1 have no decomposition relation, they are known as *relative modules.*

From this example, we can easily find that a module like M2 may have several child and descendant modules; a module like M12 may have several ancestor modules like M2 and M, but can have only one parent module like M2.

Having defined the above concepts, we are now able to define the rules for scopes of declarations of constants and type identifiers and of definitions of functions and processes.

Scope rules:

- Let M_c be a type or constant identifier declared in module M1. Then, the scope of the effectiveness of this declaration is M1 and all its descendant modules.
- Let M_c be declared or defined in both module M1 and its ancestor module M. Then M_c declared or defined in M1 has higher priority in its scope than M.

In other words, if the identifier M_c is used in module M1, it first refers to the corresponding declaration or definition in M1, if any. If not, it will refer to the first declaration in the ancestor modules sequence tracing back from M1. For example, suppose type identifier Person is used in module M122 in Figure 5.6. It first refers to the type declaration of Person in M122. If there is no such a declaration, it will trace back from M122 to search the declaration in M12, M2, and M until the declaration of Person is found. If Person is not declared in M122, M12, M2, or M, the use of Person in M122 will result in a type reference error.

However, if type Person is declared in module M1 and needs to be used in module M122, then the following reference expression must be used:

M1.Person

For example, we can declare a store variable s with type Person as follows:

s: M1.Person;

In general, if an identifier M_c is declared in module M1 and needs to be used in a relative module, say M2, of M1, the reference must be written in the format

M1.M_c

This format is also applicable to functions and processes, including their pre and postconditions. The following is a list of references to function fact used in module M2, pre and postconditions of process A, and constant identifier Age, which are all assumed to have been defined and declared in module M1:

- M1.fact(5)
- M1.**pre**-A
- M1.**post**-A
- M1.Age

5.6 Exercises

1. Answer the questions:
   a) What is a hierarchy of CDFDs?
   b) What is a hierarchy of modules?
   c) What is the relation between module hierarchy and CDFD hierarchy?
   d) What is the relation between a CDFD and its high level process in a CDFD hierarchy?
   e) What is the condition for a CDFD to be correct with respect to its high level process?
   f) What does it mean to say that module M1 is an ancestor module of M2?
   g) What does it mean to say that modules M1 and M2 are relative modules?
   h) What is the scope of a variable, type identifier, constant identifier, invariant, function, and a process?
2. Explain whether the CDFD in Figure 5.5 is structurally consistent with its high level process W. Is it possible for the CDFD to be correct with respect to process W? Explain why.

6

Explicit Specifications

As we have learned in Chapter 4, a process specification, given with pre and postconditions, is usually concise and precise in defining the functionality of the process. This kind of specification is usually suitable for defining requirements and high level design, because it allows the developer to concentrate on the relation between the inputs and outputs, with no need to think of how the relation can be implemented. However, when deriving a detailed design specification from a high level design, an algorithmic solution using sequence, choice, and iteration usually needs to be provided. Compared with program code, the detailed design specification may still be expressed in an abstract manner.

In this chapter, we introduce *statements* for writing explicit specifications. A statement performs an operation that may change the state of the process. A process specification written using statements is known as an *explicit specification*. In SOFL statements are similar to those available in high level programming languages, like Pascal, Java, and C, but their expressive power is much greater because of the use of quantified predicate expressions.

Since high level processes in a module hierarchy are defined in terms of their decompositions, there is no need to write explicit specifications for them, unless extremely necessary. Explicit specifications are mainly employed for the lowest level processes in the detailed design.

6.1 The Structure of an Explicit Specification

As we have introduced in Chapter 4, the entire structure of a process is

> **process** *ProcessName(input) output*
> **ext** *ExternalVariables*
> **pre** *PreCondition*
> **post** *PostCondition*
> **decom** *LowerLevelModuleName*

explicit *ExplicitSpecification*
comment *InformalExplanation*
end_process

The part following the keyword **explicit** is for explicit specification. The explicit specification should not coexist with the decomposition part marked by the keyword **decom**, because they both actually play the same role: describing the functionality of the process in detail. However, the explicit specification can coexist with the implicit specification written with pre and postconditions, but they may be produced at different development phases: the implicit specification is usually written in the phase of requirements analysis and/or high level design, whereas the explicit specification is usually given in detailed design. A process with only explicit specification is treated the same as that with both precondition and postcondition being **true**.

The general format of an explicit specification of a process is:

explicit
VariableDeclarations;
Statement

The *VariableDeclarations* gives the declarations of local variables within this process, and their format is the same as for declarations of local store variables in a module. *Statement* indicates an operation. Various statements are available, and they are introduced in the next section.

6.2 Assignment Statement

An assignment statement is composed of a variable and an expression, and takes the form

$$v := PE$$

The statement assigns the value resulting from the evaluation of the expression PE to the variable v, provided that the evaluation of the expression terminates. However, if the evaluation does not terminate, v will become undefined and a run time error will occur. Note that the expression PE may involve local variables, function applications, and *method invocations*; but as a method invocation usually changes the current state, the evaluation of the expression may involve state changes.

For example, the assignment statement

x := x + fact(y) + obj.m1(y)

states that variable x is updated with the result of evaluating the expression x + fact(y) + obj.m1(y). Note that x occurring in the expression denotes the

value of variable x before the execution of this assignment statement. We also assume that y is an input variable; fact(y) is a function computing the factorial of y; obj is an object of a class; and m1 is its method that yields a natural number for a given natural number y. The execution of method m1 may change the attributes of obj; therefore, the obj may actually have been changed after the execution of m1, before completing the evaluation of the entire expression. See Chapter 13 for detailed discussions about method invocations.

6.3 Sequential Statements

Sequential statements describe a series of actions, and usually contain more than one statement. The general structure of sequential statements is

```
S_1;
S_2;
...
S_n;
```

where n >= 1 and S_i(i = 1..n) are statements.

In these statements, the external variables, input variables, and output variables of the process can be used, but they should have names different from those of the local variables declared in the explicit specification.

6.4 Conditional Statements

Two conditional statements are available: **if-then** and **if-then-else**. Their formats are as follows:

(1) **if** B1 **then** S
(2) **if** B2 **then** S1 **else** S2

In the conditional statement (1), if condition B1 evaluates to true, statement S is executed; otherwise, no statement is executed. In statement (2), if condition B2 is true, statement S1 is executed; otherwise, if B2 is false, S2 is executed. For example,

if x > 10 **then** x := 20;

if x > 10 **then** x := x + 1 **else** x := x - 1;

are two legal conditional statements. The first one states that if x is greater than 10, the number 20 is assigned to variable x. The second statement states that if x is greater than 10, then x is increased by one; otherwise, if x is less than or equal to 10, x is decreased by one.

6.5 Multiple Choice Statements

Using **if-then** or **if-then-else** statements to express a behavior depending on multiple choice of values of a certain variable may lead to a deeply nested structure of statements. Such a nested structure is usually complicated and has poor readability. To resolve this problem, one way is to design a statement with a simple and clear structure, allowing the expression of behaviors based on multiple choices. Such a multiple choice statement is known as **case** statement, which, in general, has the following form:

```
case x of
ValueList_1 -> S_1;
ValueList_2 -> S_2;
       ...
ValueList_n -> S_n;
default -> S_n + 1
end_case
```

where each ValueList_i (i=1..n) is a list of concrete values of the same type as that of x; x can be either a single variable or an expression whose type matches that of the values given in ValueList_i; and S_j (j=1..n+1) are statements.

The **case** statement means that if x is equal to one of the values in ValueList_1, statement S_1 will be executed, and then the case statement will terminate; otherwise, if x is one of the values in ValueList_2, statement S_2 will be executed, and then the case statement will terminate; and so on. However, if x is different from all of the values given in ValueList_1, ValueList_2, ..., ValueList_n, statement S_n+1 will be executed as default, and then the case statement will terminate.

Note that the values in ValueList_1, ValueList_2, ..., ValueList_n should be disjoint, but even if they are not, the case statement will not involve any ambiguity in selecting one of the statements S_j (j=1..n+1) to execute, because ValueList_1, ValueList_2, ..., ValueList_n are evaluated in order, and once one of them matches x, the corresponding statement will be executed, and then the case statement will terminate. The **default** clause may not be used if ValueList_1, ValueList_2, ..., ValueList_n cover all the possible cases. However, having the **default** clause is always recommended, for it will avoid the situation of not choosing any given statement to execute. For example, consider

```
case x of
1, 2, 3 -> y := y + x + 1;
4, 5, 6 -> y := y + x + 2;
7, 8, 9 -> y := y + x + 3;
default -> y := y + x +10
end_case
```

Which one of the four assignment statements is executed depends on the value of variable x. If x is equal to 1, 2, or 3, the assign statement y := y + x + 1 is executed, and then the case statement terminates; if x is different from any given integer, the statement y := y + x +10 is executed, and the case statement terminates.

6.6 The Block Statement

A block statement in an explicit specification plays a role similar to parentheses in an expression. A block statement is used when several statements must be treated as one statement, for whatever reason. A block statement starts with keyword **begin** and ends with keyword **end**. For example, the conditional statement

```
if x > 1
then
 begin
  S_1;
  S_2;
  S_3;
 end
else
 begin
  S_4;
  S_5;
 end;
```

involves two block statements: one contains statements S_1, S_2, and S_3, and another is made up of statements S_4 and S_5. When x > 1, the first block statement is executed; otherwise, the second block statement is executed. Since each block statement is a sequence of other statements, the execution of a block statement is actually done by executing the contained statements sequentially.

6.7 The While Statement

A **while** statement describes an iteration of executions, and takes the form

```
while B do
 S
```

When condition B evaluates to true, S is executed repeatedly until B becomes false. To ensure the termination of the **while** statement, there must be some variables involved in S and B that control the iteration. For example,

```
f = 1;
n = 10;
while n > 1 do
begin
f = f * n;
n = n - 1;
end;
```

The **while** statement computes the factorial of natural number 10, and the result is held in variable f. The variable n is used to control the termination of the while statement.

Since the readability and verification of **while** statements are usually difficult, and writing them usually involves detailed consideration of algorithms, one should avoid using them as much as possible in explicit specifications. Instead, recursive functions should be considered whenever an iteration is needed. For example, if we make use of function application fact(10), where fact is assumed to have been defined before as a recursive function computing factorials of a natural number, the **while** statement given above is simplified as the following statement:

```
f = fact(10);
```

6.8 Method Invocation

Although the issue of invoking a method of an object should be discussed after the introduction of "class" and "object," we need to emphasize here that invoking a method can be part of an explicit specification of a process. If this topic feels premature, you can skip this section. The material will be discussed in detail in Chapter 13.

Let obj be an object of the class Obj, which is treated as a user-defined type, and m1 be a method defined in class Obj. If m1 does not yield any output value, the method invocation

```
obj.m1();
```

can be used as an independent statement in explicit specifications. However, if m1 returns an output value, it can be used in any place appropriate (e.g., the expression involved in an assignment statement); see Chapter 13 for more details.

6.9 Input and Output Statements

Sometimes it may be necessary to express the idea of either reading values from outside the system under construction (e.g., input device like keyboard) or writing values to a device outside the system (e.g., output device like printer and display). In this case, we need appropriate input and output statements. Since explicit specifications are still intended to be an abstraction of the ultimate program, the input and output statements are designed to facilitate the specification of what to input or output, without caring about how the input and output are done. The format of input and output needs to be decided during the implementation of the system.

The general form of the input statement is

read($x\_1, x\_2, ..., x\_n$)

The **read** statement reads values from the input device sequentially into variables $x\_1, x\_2, ..., x\_n$, respectively. The types of these variables may vary, but the type of the value to be read must be kept the same as that of the variable to which the value is read.

The output statement takes the following form:

write($e\_1, e\_2, ..., e\_m$)

The write statement writes the results of expressions $e\_1, e\_2, ..., e\_m$ sequentially to the output device. The types of these expressions may vary as well. For example, the statement

read(x, y)

reads two values sequentially from the input device to variables x and y, respectively. While the statement

write("The result is", x + y, '!')

writes "The result is," the result of x + y, and the character '!,' in turn, to the output device.

6.10 Example

Let us take the process Check_Password of the ATM specification given in Chapter 4 as an example to illustrate how to write an explicit specification for a process. As given before, the implicit specification of the process is as follows:

process Check_Password(card_id: **nat**, sel: **bool**, pass: **nat**)
                         account1: Account | pr_meg: **string** |
                         account2: Account
ext rd account_file /*The type of this variable is omitted because
                    this external variable has been declared in
                    the **var** section. */
post (**exists!**[x: account_file] |
              x.account_no = card_id **and**
              x.password = pass **and**
              (sel = **false and** account1 = x **or**
               sel = **true and** account2 = x)
       or
       not (**exists!**[x: account_file] | x.account_no = card_id **and**
                            x.password = pass) **and**
          pr_meg = "Reenter your password or insert the correct card"
    comment
If the input card_id and pass are correct with respect to the existing information
in account_file, then if sel is false, the account information is passed to the output
account1; otherwise, the account information is passed to the ouput account2.
However, if one of card_id and pass is incorrect, a prompt message pr_meg is
produced.
    end_process;

An explicit specification implementing this implicit specification is given
as follows:
process Check_Password(card_id: **nat**, sel: **bool**, pass: **nat**)
                         account1: Account | pr_meg: **string** |
                         account2: Account
ext rd #account_file: **set of** Account;
explicit
begin
   account1 := **get**({x | x: account | x.account_no = card_id **and**
                              x.password = pass});
   if account1 = **nil**
   then pr_meg = "Reenter your pass or insert the correct card"
   else if sel = **true**
        then
         begin
           account2 := account1;
           account1 := **nil**;
         end
      else account2 := **nil**;
end
comment
...
end_process;

The explicit specification consists of a block statement that contains two statements: an assignment statement followed by a conditional statement. The assignment statement is intended to assign the account in account_file whose number and password are the same as required by the input value to the variable account1. If such an account does not exist, that is, account1 = **nil**, the prompt message pr_meg is given. However, if the account does exist, and variable sel is equal to true, account2 is updated with account1, and account1 is set as **nil**; otherwise, account2 is set as **nil**. In this specification, the operator **get** defined on set types is applied to obtain an element (no matter which one) in the set defined by the set comprehension **get**({x | x: account | x.account_no = card_id **and** x.password = pass}). Detailed discussions on set comprehension and the operator **get** are given in Chapter 8.

6.11 Exercises

Write explicit specifications for the following processes of the ATM given in Chapter 4.

1. a) Receive_Command
    b) Withdraw
    c) Show_Balance

7

Basic Data Types

Data types are essential for specifications because they provide a notation for defining data structures used in specifications. From this chapter, through Chapter 13, we introduce all the data types available in SOFL. Data types are divided into two categories: *built-in types* and *user-defined types*. The built-in types are further divided into basic types and compound types. The compound types include set types, sequence types, composite types, map types, product types, and union types. The user-defined types are the types that can be defined by the specification writers for constructing well-structured, maintainable, and reusable specifications. The user-defined type is known as *class*. At the end of each chapter, examples are given to show how the introduced types are used to define data structures for process specifications.

In this chapter, we focus on the basic types, while from the next chapter through Chapter 12, we introduce set types, sequence and string types, composite types, map types, product types, and union types, in that order. Classes and their instantiations are discussed in Chapter 13.

The basic types include numeric types, boolean type, character type, and enumeration types. Since boolean type has been introduced in Chapter 2, we will just give a brief description of it.

7.1 The Numeric Types

Four numeric types are employed in SOFL; they are natural numbers including zero, natural numbers, integers, and real numbers. These types are denoted by the symbols **nat0**, **nat**, **int**, and **real**, respectively, and their values are already explained in Chapter 3.

Several *arithmetic operators* and *relational operators* are provided for computing numeric values. The arithmetic operators and their names, as well as types, are given in Table 7.1. In the table, product types, such as **real * real**, are used in defining the types of the operators. The reader who is not familiar with the concept of product type can refer to Chapter 10 for details.

Table 7.1. Arithmetic operators

Operator	Name	Type
- x	Unary minus	**real** –> **real**
abs(x)	Absolute value	**real** –> **real**
floor(x)	Floor	**real** –> **int**
x + y	Addition	**real** * **real** –> **real**
x - y	Subtraction	**real** * **real** –> **real**
x * y	Multiplication	**real** * **real** –> **real**
x / y	Division	**real** * **real** –> **real**
x **div** y	Integer division	**int** * **int** –> **int**
x **rem** y	Remainder	**int** * **int** –> **nat0**
x **mod** y	Modulus	**nat0** * **nat0** –> **nat0**
x ** y	Power	**real** * **real** –> **real**

Each operator is a function that yields a single value when applied to its arguments. It is worth noting that every operator with parameters of "super-types" can apply to arguments of "subtypes." For example, an operator with parameters of type **real** can apply to arguments of types **int**, **nat**, and **nat0**; parameters of **int** can apply to arguments of **nat**; and parameters of **nat** can apply to arguments of **nat0**. Since most of these operators are commonly used in fundamental mathematics, there is no need to explain their semantics formally here. Instead, some examples may be more helpful.

For example, let $x = 9$, $y = 4.5$, $z = 3.14$, $a = -4$, and $b = 3$. Then, we apply these operators and get the following results:

- z = - 3.14
abs(a) = 4
floor(y) = 4
x + z = 12.14
x - y = 4.5
a * b = - 12
x / y = 2.0
a **div** b = -1
a **rem** b = 1
x **mod** b = 0
x ** b = 729

The relational operators over numeric types are given in Table 7.2. Each relational operator is a predicate that takes some arguments and yields a truth value. Except the less-between and less-equal-between operators, all the other relational operators have been used in previous chapters, and are supposed to be familiar to the reader. x < y < z evaluates to **true** if y is greater than x but less than z; otherwise, it yields **false**. x <= y <= z evaluates to **true** if y is

Table 7.2. Relational operators

Operator	Name	Type
x < y	Less than	real * real –> bool
x > y	Greater than	real * real –> bool
x <= y	Less or equal	real * real –> bool
x >= y	Greater or equal	real * real –> bool
x < y < z	Less-between	real * real * real –> bool
x <= y <= z	Less-equal-between	real * real * real –> bool
x = y	Equal	real * real –> bool
x <> y	Not equal	real * real –> bool

greater than or equal to x, but less than or equal to z; otherwise, it evaluates to **false**.

For example, let x = 9, y = 4.5, and z = 12.5. Then,

x > y <=> **true**
x < y <=> **false**
x <= y <=> **false**
x >= y <=> **true**
y < x < z <=> **true**
x < y < z <=> **false**
y <= y <= z <=> **true**
x = x <=> **true**
x <> z <=> **true**

In fact, x < y < z is equivalent to the conjunction x < y **and** y < z, and x <= y <= z is equivalent to the conjunction x <= y **and** y <= z.

7.2 The Character Type

Character is the atomic unit for constructing identifiers (for names, variables, types, constants), operators of types, and delimiters for separating different parts in a specification. The character type contains all the characters of the SOFL character set, defined in Table 7.3. The type is denoted by the keyword

char

and each character value is written in the form

'x'

where x is a single element of the SOFL character set. For example, the following is a list of legal characters:

Table 7.3. SOFL character set

| English letters: |
| a b c d e f g h i j k l m n o p q r s t u v w x y z |
| A B C D E F G H I J K L M N O P Q R S T U V W X Y Z |
| Other characters: |
| , . : ; * + - / _ ˜ \| \ ( ) [ ] { } @ ˆ ' ' & % $ # " ! < > = ? |
| Newline: |
| White space: |

'a'
'B'
'\|'
')'
'.'
'@'
'7'

The only operators available on the type **char** are the two relational operators "=" and "<>". Thus, characters can be compared with each other for their equality. For example,

'a' = 'A' <=> **false**
'a' = 'a' <=> **true**
'8' = '=' <=> **false**

7.3 The Enumeration Types

An enumeration type is a finite set of special values, usually with the feature of describing a systematic phenomenon. For example, the days of a week can be modeled as an enumeration type Week:

Week = {<Monday>, <Tuesday>, <Wednesday>, <Thursday>, <Friday>,
        <Saturday>, <Sunday>}

Each value in an enumeration type is written in the form:

<x>

where x is a string of SOFL characters.

Although each value of an enumeration type is enclosed by a pair of angle brackets, the brackets are just part of the syntax to help distinguish values of enumeration types from normal string values (see Chapter 9 for the detailed

discussion of the **string** type); they do not add any additional meaning to the values. For example, you should understand <Monday> in the same way as you understand the notion Monday.

If we declare a variable weekday with the type Week as

weekday: Week;

then the variable can take any value of the type, that is, weekday can take <Monday>, or <Tuesday>, or <Wednesday>, and so on, as its value.

The only operators over an enumeration type are equality and inequality, allowing the comparison between values of an enumeration type. For example, suppose we declare

x, y, z: Week;

and let x = <Monday>, y = <Wednesday>, and z = <Monday>, then:

x = y <=> **false**
x = z <=> **true**
y <> z <=> **true**
x <> z <=> **false**

7.4 The Boolean Type

The boolean type is denoted by the symbol: **bool**, and it contains only two values: **true** and **false**, as we have already learned from the previous chapters. The only thing we would like to mention about the boolean type here is the additional operators: "=" and "<>".

If there is no confusion, the equality symbol "=" can be used in the same way as <=> for boolean values (i.e., truth values). The inequality symbol "<>" can be applied to check whether two boolean values are not equivalent. For example, suppose we declare

p, q, r: **bool**;

and let p = **true**, q = **false**, and r = **true**. Then,

p = r <=> **true**
p <> q <=> **true**
p <> r <=> **false**.

There is a rule on the priority of applying relational operators and logical operators in predicate expressions: relational operators always have higher priority than logical operators. For example, when both a relational operator,

= or <>, and the logical operator, <=>, are used for variables (including boolean variables), the priority of = or <> is always higher than that of <=>. Thus, when evaluating the expression p = r <=> **true**, the relation p = r must evaluate first, followed by the entire expression.

7.5 An Example

Let us specify a simple process, reporting fares of railway tickets for different kinds of passengers, as an example to illustrate the use of the basic types in process specifications. Assume that the fare for *student* is 25 percent less than the normal fare, and the ticket for the *pensioner* is 30 percent less than the normal fare. All the other people are treated as ordinary passengers. The process is then specified as follows:

```
type
Passenger = {<STUDENT>, <ORDINARY>, <PENSIONER>};
process Tell_Fare(passenger: Passenger)
                fare: real
ext rd normal_fare: real
post fare = case passenger of
        <STUDENT> -> normal_fare - 0.25 * normal_fare;
        <ORDINARY> -> normal_fare;
        <PENSIONER> -> normal_fare - 0.30 * normal_fare
        end_case
end_process;
```

The input of this process is the type of the passenger: student, ordinary passenger, or pensioner. The output is the fare of the ticket the passenger needs to pay. The normal fare of the ticket is treated as a **rd** type external variable. In the postcondition, a case expression is used to define the output fare based on the input passenger and the railway company's ticket discount policy.

7.6 Exercises

1. Let x = 12, y = 9.8, z = 2, and a = -20. Evaluate the following expressions:
   a) - z
   b) **abs(a)**
   c) **floor(y)**
   d) x + z
   e) x - y
   f) a * z
   g) x / y
   h) a **div** z

i) a **rem** x

j) x **mod** z

k) x ** z

2. Let x = 20, y = 5.5, z = 'd', and a = **true**. Evaluate the following expressions:

a) 'a' = z

b) ')' <> z

c) x >= y

d) x < y <= y

e) a = **false**

f) a <> **true**

3. Assume that the courses to teach on weekdays are as follows: "Software Engineering" on Monday, "Program Design" on Tuesday, "Discrete Mathematics" on Wednesday, "Programming Language" on Thursday, and "Formal Engineering Methods" on Friday. Write a formal specification for the process that gives the corresponding course title for an input weekday.

8

The Set Types

Computation by computer is not limited only to numeric calculations. In fact, computation has a more broad meaning: manipulation of data or information by algorithms. Many commercial program systems and those in the public domain actually have little to do with numeric calculation; their essential functionalities are closely related to dealing with data structures. For example, searching for a data item in a file, or sorting students' examination results in a list, can be regarded as such kinds of programs.

The set types are one of the compound types available in SOFL, and usually used for the abstraction of data items that have a collection of elements. A set type contains a collection of set values with the same feature (e.g., their elements are the values of the same type) and a group of operators. The collection of set values offers a range of set values that can be taken by a variable of the set type, while the operators are used to manipulate set values.

In this chapter, we first explain the concept of set, and then introduce the *set type constructor*, by which a specific set type can be constructed. Furthermore, all the operators on set types are discussed in Section 8.3, and examples of applying sets for data abstraction in process specifications are given to explain how set data structures can be used for data abstraction in specifications.

8.1 What Is a Set

A set is an unordered collection of distinct objects where each object is known as an *element* of the set. Since computers can deal with only finite sets, we require that any set (value) of a set type be finite (i.e., it contains finite number of elements). For example, a school class is a set of students; a car park is a set of cars; a library is a set of books; and so on. Set values are written as a list of their elements, separated by commas, and enclosed by braces. For example, the following are some set values:

(1) {5, 9, 10}
(2) {"John", "Chris", "David", "Jeff"}
(3) {"Java", "Pascal", "C", "C++", "Fortran"}

Set (1) is a set of three natural numbers; set (2) denotes a set of four people's names; and set (3) shows a set of programming languages.

An essential property a set value has is that its elements are unordered, that is, changing the order of the elements does not change the value of the set. Thus,

{5, 9, 10} = {9, 5, 10}, and
{"John", "Chris", "David", "Jeff"} = {"David", "Chris", "Jeff", "John"}

Another important property of a set is that there is no duplication of its elements. For example,

{5, 9, 10, 5}

is an illegal set value because element 5 appears twice. It is important to remember these two properties when using set values to model data items and when reading a specification involving set values.

8.2 Set Type Declaration

A set type is declared by applying the set type constructor to an element type. The set type constructor is

set of

Applying this constructor to a specific element type, say T, yields a specific set type. Let A denote this type, then we can write:

A = **set of** T

which represents a set type in which each set value is a collection of elements of type T. Formally, A is a power set of T, i.e.

$A = \{x \mid \textbf{subset}(x, T)\}$.

where **subset**(x, T) means that x is a subset of T (this operator is defined in Section 8.3.2).

For example, let T be the enumeration type:

$T = \{<\text{STUDENT}>, <\text{ORDINARY}>, <\text{PENSIONER}>\}$

Then, a set type ST is declared as

ST = **set of** T

Thus, ST is a power set of T:

ST = {{ }, {<STUDENT>}, {<ORDINARY>}, {<PENSIONER>},
    {<STUDENT>, <ORDINARY>},
    {<ORDINARY>, <PENSIONER>},
    {<STUDENT>, <PENSIONER>},
    {<STUDENT>, <ORDINARY>, <PENSIONER>}}

where { } denotes the empty set.

In a specification, a declaration of variable x with type ST can be given as

x: ST

This allows variable x to take any value of type ST. For instance, x can take the following set values:

x = {<STUDENT>}
x = {<ORDINARY>, <PENSIONER>}
x = { }
x = {<STUDENT>, <ORDINARY>, <PENSIONER>}

8.3 Constructors and Operators on Sets

Given a specific set type, set values of the type can be constructed in two ways. One is by using the *constructors*, and another is by applying the *operators* to existing set values. A constructor of the set type is a special operator that constitutes a set value from the elements of an element type, while a normal operator is used to generate a set value based on existing set values. Since the term "set" is the synonym of "set value," we use them alternately as necessary for the convenience of discussions in this chapter.

8.3.1 Constructors

There are two sets constructors: *set enumeration* and *set comprehension*. A set enumeration has the format

{e_1, e_2, ..., e_n}

where e_i (i=1..n) are the elements of the set {e_1, e_2, ..., e_n}. For example, a set of integers is

{5, 9, 10, 50}

A set comprehension defines a set containing all the elements satisfying some property. The general form of a set comprehension is

$$\{e(x\_1, x\_2, ..., x\_n) \mid x\_1: T\_1, x\_2: T\_2, ..., x\_n: T\_n \&$$
$$P(x\_1, x\_2, ..., x\_n)\}$$

where $n \geq 1$.

The set comprehension defines a collection of values resulting from evaluating the expression $e(x\_1, x\_2, ..., x\_n)$ $(n \geq 1)$ under the condition that the involved variables $x\_1, x\_2, ..., x\_n$ take values from sets (or types) T_1, $T\_2, ..., T\_n$, respectively, and satisfy property $P(x\_1, x\_2, ..., x\_n)$. If it is obvious or unnecessary, the bindings in the set comprehension can be omitted. Thus, we may use another form of set comprehension:

$$\{e(x\_1, x\_2, ..., x\_n) \mid P(x\_1, x\_2, ..., x\_n)\}$$

Some examples are given below to illustrate the use of set comprehensions.

$$\{x \mid x: \textbf{nat} \& 1 < x < 5\} = \{2, 3, 4\}$$
$$\{y \mid y: \textbf{nat0} \& y <= 5\} = \{0, 1, 2, 3, 4, 5\}$$
$$\{x + y \mid x: \textbf{nat0}, y: \textbf{nat0} \& 1 < x + y < 8\} = \{2, 3, 4, 5, 6, 7\}$$
$$\{i \mid i: \textbf{nat0} \& 9 < i < 4\} = \{ \ \}$$
$$\{i \mid i \textbf{ inset nat and } i < 5\} = \{1, 2, 3, 4\}$$

We can also use the following special notation to represent a set containing an interval of integers:

$$\{i, ..., k\} = \{j \mid j: \textbf{int} \& i <= j <= k\}$$

Thus,

$$\{1, ..., 5\} = \{1, 2, 3, 4, 5\}$$
$$\{-2, ..., 2\} = \{-2, -1, 0, 1, 2\}$$

8.3.2 Operators

In addition to the set constructors, there are also operators for manipulating set values. Given the element type T, all the built-in operators on set types are discussed below one by one.

Membership

The operator for determining whether a value is a member of a set is **inset**.

inset: T * **set of** T –> **bool**

The expression x **inset** s evaluates to true if x is a member of set s; otherwise, it yields false. For example,

> 7 **inset** {4, 5, 7, 9} <=> **true**
> 3 **inset** {4, 5, 7, 9} <=> **false**

Non-membership

The operator for determining if a value is not a member of a set is **notin**.

> **notin**: T * **set of** T –> **bool**

If x is not a member of s, the expression x **notin** s evaluates to true; otherwise, it evaluates to false. Note that the non-membership operator is opposite to the membership operator in determining whether a value is a member of a set. For example,

> 7 **notin** {4, 5, 7, 9} <=> **false**
> 3 **notin** {4, 5, 7, 9} <=> **true**

Cardinality

The cardinality of a set means the number of the elements in the set. The cardinality operator is **card**.

> **card**: **set of** T –> **nat0**
> **card**(s) == the number of elements in s

where == means "is defined as," the same as for defining a function introduced in Chapter 4.

When applying the operator **card** to a set value, say x, we are required to use parentheses to enclose the argument. For example,

> **card**({5, 7, 9}) = 3
> **card**({'h', 'o', 's', 'e', 'i'}) = 5

Equality and inequality

Two sets can be compared to determine if they are identical or not. Set s1 is equal to s2, that is, they are identical, if they have exactly the same members; otherwise, they are not identical. The operators for equality and inequality of sets are = and <>, the same as for numeric values.

> =: **set of** T * **set of** T –> **bool**

s1 = s2 == **forall**[x: s1] | x **inset** s2 **and** card(s1) = card(s2)
<>: set of T * set of T –> bool
s1 <> s2 == (**exists**[x: s1] | x **notin** s2) **or** (**exists**[x: s2] | x **notin** s1)

For example,

{5, 15, 25} = { 5, 15, 25} <=> **true**
{5, 15, 25} <> {5, 20, 30} <=> **true**

Subset

A set s1 is said to be a subset of another set s2 if the members of s1 are all the members of s2. The operator is **subset**.

subset: set of T * set of T –> bool
subset(s1, s2) == **forall**[x: s1] | x **inset** s2

If s1 is a subset of s2, **subset**(s1, s2) evaluates to true; otherwise, it evaluates to false. Let s1 = {5, 15, 25}, s2 = {5, 10, 15, 20, 25, 30}. Then,

subset(s1, s2) <=> **true**
subset(s2, s1) <=> **false**
subset({ }, s1) <=> **true**
subset(s1, s1) <=> **true**

The third expression shows that the empty set is a subset of s1. In fact, the empty set is a subset of any set. The fourth expression states that set s1 is a subset of itself, which is also true of any other set.

Proper subset

The set s1 is a proper subset of the set s2 if the members of s1 are all the members of s2 and s1 is not equal to s2. The operator for proper subset is **psubset**.

psubset: set of T * set of T –> bool
psubset(s1, s2) == **subset**(s1, s2) **and** s1 <> s2

For example, let s1 = {5, 15, 25} and s2 = {5, 10, 15, 25, 30}. Then,

psubset(s1, s2) <=> **true**
psubset(s1, s1) <=> **false**
psubset(s2, s1) <=> **false**
psubset({ }, s1) <=> **true**

The empty set { } is a proper subset of any set except the empty set itself.

Member access

The member access operator is designed for obtaining a member from a set. The operator is **get**, and it is defined as follows:

> **get**: **set of** T $->$ T
> **get**(s) $==$ **if** s $<>$ { } **then** x **else nil**

where x **inset** s.

Note that the application of the **get** operator to a set yields a member of the set in a nondeterministic manner. If the set is empty, the application becomes undefined (i.e., it yields **nil**). It is also important to bear in mind that **get**(s) returns a member of set s, but does not change s. For example, assume s = {5, 15, 25}; then,

> **get**(s) $= 5$ or
> **get**(s) $= 15$ or
> **get**(s) $= 25$

and s still remains the same as before: s = {5, 15, 25}.

Union

The union of sets is an operation to merge two sets into one, that is, to join together their members to form another set. The operator is **union**, and it is defined as

> **union**: **set of** T ***** **set of** T $->$ **set of** T
> **union**(s1, s2) $==$ {x | x **inset** s1 **or** x **inset** s2}

Since a set must have no duplication of members, the union of sets must also be performed in the way that ensures this property. Consider the following examples:

> **union**({5, 15, 25}, {15, 20, 25, 30}) $=$ {5, 15, 25, 20, 30}
> **union**({15, 20, 25, 30}, {5, 15, 25}) $=$ {15, 20, 25, 30, 5}

A simple way to obtain the result of the operation **union**(s1, s2) is to include all the members of s1 in the resulting set and then extend it by adding the members of s2 that do not belong to s1, as shown in the above examples.

The union operator is commutative. Thus, **union**(s1, s2) $=$ **union**(s2, s1). It is also associative, that is, **union**(s1, **union**(s2, s3)) $=$ **union**(**union**(s1, s2), s3). Due to these properties, the operator **union** can be extended to deal with more than two sets:

union: **set of** T * **set of** T * ... * **set of** T –> **set of** T
union(s1, s2, ..., sn) == {x | x **inset** s1 **or** x **inset** s2 **or** ... **or** x **inset** sn}

The use of this extended **union** operator is an effective way to shorten expressions involving the union operation of many sets. This can be understood by seeing the following two expressions as equivalent:

union(s1, **union**(s2, **union**(s3, ...)))
union(s1, s2, ..., sn)

Intersection

The intersection of two sets yields a set that contains the common members of the two sets. The operator for intersection of sets is **inter**, and is defined as

inter: **set of** T * **set of** T –> **set of** T
inter(s1, s2) == {x | x **inset** s1 **and** x **inset** s2}

For example, let s1 = {5, 7, 9}, s2 = {7, 10, 9, 15}, and s3 = {8, 5, 20}. Then,

inter(s1, s2) = {7, 9}
inter(s1, s3) = {5}
inter(s2, s3) = { }

The properties of commutativity and associativity also hold for the **inter** operator. That is, **inter**(s1, s2) = **inter**(s2, s1), and **inter**(s1, **inter**(s2, s3)) = **inter**(**inter**(s1, s2), s3). Taking the same approach as that of extending the **union** operator, we extend the **inter** operator as follows:

inter: **set of** T * **set of** T * ... * **set of** T –> **set of** T
inter(s1, s2, ..., sn) == {x | x **inset** s1 **and** x **inset** s2 **and** ... **and** x **inset** sn}

Difference

The difference between two sets is an operation that yields another set. The operator for the difference operation is **diff**, and it is defined as

diff: **set of** T * **set of** T –> **set of** T
diff(s1, s2) == {x | x **inset** s1 **and** x **notin** s2}

For example, let s1 = {5, 7, 9}, s2 = {7, 10, 9, 15}, and s3 = {8, 12}. Then,

diff(s1, s2) = {5}

diff(s1, s3) = {5, 7, 9}
diff(s2, s1) = {10, 15}
diff(s1, { }) = s1

Distributed union

A set can be a set of sets, and the distributed union of such a set is an operation that obtains the union of all the member sets of the set. The operator for distributed union operation is **dunion**. For example, suppose s1 = {{5, 10, 15}, {5, 10, 15, 25}, {10, 25, 35}} is a set of the three sets: {5, 10, 15}, {5, 10, 15, 25}, and {10, 25, 35}; then, the distributed union of s1 is

dunion(s1) = **union**({5, 10, 15}, {5, 10, 15, 25}, {10, 25, 35}}
            = {5, 10, 15, 25, 35}

Formally, the distributed union operator is defined as

dunion: **set of set of** T –> **set of** T
dunion(s) == **union**(s1, s2, ..., sn)

where s = {s1, s2, ..., sn}.

Distributed intersection

Similarly to the distributed union operator, the distributed intersection operator also applies to a set of sets. The operator is **dinter**, and it is defined as

dinter: **set of set of** T –> **set of** T
dinter(s) == **inter**(s1, s2, ..., sn)

where s = {s1, s2, ..., sn}.
    For example, let s1 = {{5, 10, 15}, {5, 10, 15, 25}, {10, 25, 35}}. Then,

dinter(s1) = **inter**({5, 10, 15}, {5, 10, 15, 25}, {10, 25, 35}}
            = {10}

Power set

Given a set, we can apply the operator **power** to yield its power set that contains all the subsets of the set, including the empty set. The power set operator is defined as follows:

power: **set of** T –> **set of set of** T
power(s) == { s1 | **subset**(s1, s) }

For example, let s = {5, 15, 25}. Then,

power(s) = {{ }, {5}, {15}, {25}, {5, 15}, {15, 25}, {5, 25}, {5, 15, 25}}

8.4 Specification with Set Types

The importance of learning the set notation introduced in this chapter is to understand how it can be used for process modeling and specification. A process usually deals with data items, and these data items may be defined as sets. The most interesting question is whether it is appropriate to declare the data items as sets. To answer this question, we must examine the nature of the data items to see whether they are intended to represent *unordered collections* of *distinct* objects. If so, choosing sets as the abstraction of the data items would be a right decision.

In this section, we use the example of an *email (electronic mail) address book* to illustrate how the set notation is used for process specifications. Usually, there is no benefit either in recording duplicated email addresses in the address book or in defining a specific order in which the email addresses are organized, as long as the addresses can be easily managed (e.g., through find, add, and delete operations). Therefore, it is sufficient to abstract the email address book as a set of email addresses. An email address is usually a string of characters, but for now there is no point to go into that detail. So we declare Email as a given type:

Email = **given**;

Based on this type declaration, we declare a state variable email_book and three processes to manipulate the email addresses contained in the book. The processes are add, find, and delete. All of these components are defined in the module Email_Address_Book.

module Email_Address_Book;

type

Email = **given**;

var

email_book: **set of** Email;

behav: CDFD_8.1;

process Find(e: Email) r: **bool**

ext rd email_book
post r = (e **inset** email_book)
end_process;

process Add(e: Email)
ext wr email_book
post email_book = **union**(~email_book, {e})
end_process;

process Delete(e: Email)
ext wr email_book
post email_book = **diff**(~email_book, {e})
end_process;

end_module;

The process Find checks whether a given email address e is already included in email_book. If it is, the process assigns **true** to the output variable r; otherwise it assigns **false** to r. Since the checking can be done for any given email address, no specific precondition is required by the process.

The process Add takes an email e and adds it to the email address book email_book. This function is described by using a set union operation: email_book = **union**(~email_book, {e}), in the postcondition. Since the operator **union** preserves the property of a set to disallow duplication of elements in the set, element e will not be added to the set ~email_book if e does not belong to the set before the union operation. For this reason, the precondition of this process is defined as **true**, imposing no specific constraint on the input variable e and the external variable email_book.

The elimination of an email address from email_book is done by means of the process Delete. It is specifically defined by an application of the set difference operator **diff** in the expression email_book = **diff**(~email_book, {e}) of the postcondition. Since operator **diff** properly deals with both situations, when e belongs and does not belong to ~email_book, no specific requirement for e to be a member of ~email_book in the precondition is given.

The overall behavior of this module is depicted by the CDFD in Figure 8.1. If the input data flow e of process Find is available, the process is executed and its output r is generated. If the input e of process Add is available, Add is executed and the store email_book is updated. Furthermore, if the input e of process Delete is available, the process is executed and the store email_book is updated. In the CDFD, it is assumed that the input data flows of the three processes are exclusively available. Thus, only one of processes Add, Find, and Delete can be executed at any time.
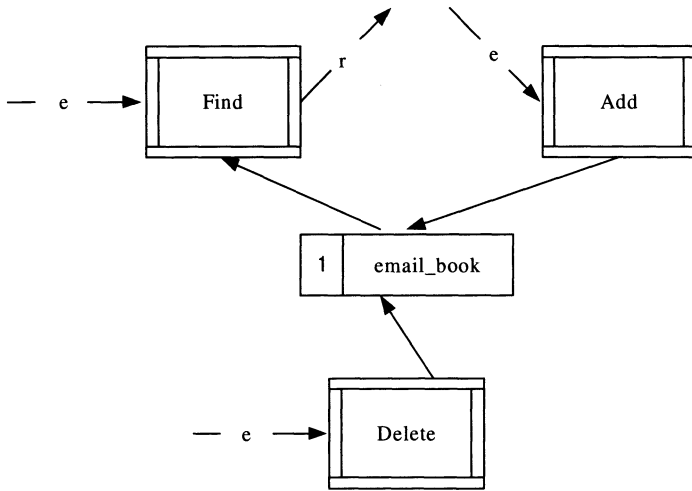
Fig. 8.1. The CDFD for the module Email_Address_Book

8.5 Exercises

1. Given a set T = {5, 8, 9}, define a set type based on T, and list all the possible set values in the type.

2. Let T = {5, 8, 9}. Evaluate the following set comprehensions:
   a) {x | x: **nat** & x < 8}
   b) {y | y: **nat0** & y <= 3}
   c) {x - y | x: **int**, y: **int** & -2 < x < 3 **and** -1 < y < 2}
   d) {i | i: **set of** T & **card**(i) < 3 **and forall**[x, y: i] | x + y <= 13}

3. Let s1 = {5, 15, 25}, s2 = {15, 30, 50}, s3 = {30, 2, 8}, and s = {s1, s2, s3}. Evaluate the following expressions:
   a) **card(s1)**
   b) **card(s)**
   c) **union(s1, s2)**
   d) **diff(s2, s3)**
   e) **inter(union(s2, s3), s1)**
   f) **dunion(s)**
   g) **dinter(s)**
   h) **inter(union(s1, s3), diff(s2, union(s1, s3)))**

4. Write set comprehensions for the following sets:
   a) a set of natural numbers whose elements are all smaller than 10.
   b) a set of integers whose elements are all greater than 0 and smaller than 10 and cannot be divided by 3.
   c) a set of prime numbers.

5. Construct a module to model a telephone book containing a set of telephone numbers. The necessary processes are Add, Find, Delete, and Update. The process Add adds a new telephone number to the book; Find determines whether a given telephone number is available or not in the book; Delete eliminates a given telephone number from the book; and Update replaces an old telephone number with a new number in the book.

6. Write a specification for a process Merge. The process takes two groups of students, and merges them into one group. Since the merged group will be lectured by a different professor, the students from both groups may drop from the merged group (but exactly which students will drop is not known).

9

The Sequence and String Types

The set types have provided us with a powerful tool for the abstraction of data items. However, they are not sufficient for modeling data items for various requirements. Sometimes we may need to clearly emphasize the order of the elements in a set, such as a list of countries with the number of gold medals obtained at the 2000 Sydney Olympics. The position of each country in the list cannot be changed because this would otherwise probably alter the number of the gold medals obtained by those countries. Some other times we may need to record duplicate elements in a set, each duplicate element denoting a different object. For example, it is quite possible that a class of undergraduates has two or more students with the same name, so defining such a class as a set is obviously inappropriate. If one wants to represent the ages of all the students in a class as a set corresponding to the name set of the class, one must be careful in using a set type, because the class is very likely to have more than one student with the same age.

To model data items with a set of values whose order is important and whose duplications are possible, we introduce another kind of data type, *sequence type*, in this chapter. A special kind of sequence type, the *string* type, is also discussed.

9.1 What Is a Sequence

A sequence is an *ordered collection of objects that allows duplications of objects.* As with sets, the objects are known as *elements* of the sequence. As mentioned previously, there are two important differences between sequences and sets. The order of the elements of a sequence is important whereas the order of the elements of a set is not significant. Thus, changing the order of the elements of a sequence usually changes the sequence to a different one, whereas the change of the order of elements in a set does not change the set. Another difference is that a sequence allows duplicated elements whereas a set

does not. For the same reason as for sets, we only deal with finite sequences (i.e., the sequences containing a finite number of elements).

A sequence value is represented by a list of elements contained within square brackets, [ ]. The elements are all from the same type, called *element type*. For example, the following are some sequences:

(1) [5, 15, 15, 5, 35]
(2) ['u', 'n', 'i', 'v', 'e', 'r', 's', 'i', 't', 'y']
(3) [20.5, 40.5, 85.5]

Sequence (1) provides a group of natural numbers; sequence (2) is a group of characters; and sequence (3) gives a list of real numbers. A sequence with no element is called *empty sequence*. The empty sequence has the form

[ ]

Since sequences composed of characters like sequence (2) are often used to represent names, addresses, and other texts, it would appear more natural to write them as a string of characters, such as "university", rather than as a sequence. We classify all the sequences composed of characters into a new type known as **string** (but this does not disallow us to use sequence of characters in sequence notation). A string value is a list of characters in double quotes, such as:

"university"
"sofl@yahoo.ac.jp"
"Formal Engineering Methods"

All the operators on sequences, to be introduced in this chapter, are applicable to string values in the same manner. So, we will just apply those operators to string values in examples as necessary, without additional explanations.

9.2 Sequence Type Declarations

A sequence type is declared by applying the sequence type constructor

seq of

to a specific element type. For example,

seq of nat

forms a sequence type in which each sequence is constituted by a collection of natural numbers. A new type identifier can be declared using the sequence type. For example,

Ages = seq of nat

Thus, Ages can be used as the same type as **seq of nat**, containing all the sequence values whose elements are natural numbers. Using this type identifier, the variable student_ages can then be declared as

student_ages: Ages;

Thus, the variable student_ages may take any value in the type Ages.

9.3 Constructors and Operators on Sequences

As with the set notation, sequences can be created by applying either sequence constructors or operators. In this section, we discuss all the constructors and operators on sequence types.

9.3.1 Constructors

A constructor is a special operator that allows us to form sequences from element types. There are two constructors: *sequence enumeration* and *sequence comprehension*. A sequence enumeration has the format

[a_1, a_2, ..., a_n]

where a_i (i=1..n) are the elements of the sequence. For example,

[5, 9, 8, 9, 5]

is a sequence of natural numbers. The order and occurrence of the elements are significant. Thus,

[5, 9] <> [9, 5]

and

[5, 9, 5] <> [5, 9]

When forming a sequence, it is important to ensure that all the elements are the values of the same type. Thus, we should be careful in constructing the sequence,

[5, 'a', "university", 20.05]

unless a sequence of *union type* (to be introduced in Chapter 12) is desired.

A sequence comprehension is similar to a set comprehension, but since the order of elements is significant and the duplicated occurrences of elements are possible, sequence comprehensions need to be constructed with caution. A sequence comprehension takes the format:

$$[e(x\_1, x\_2, ..., x\_n) \mid x\_1: T\_1, x\_2: T\_2, ..., x\_n: T\_n \ \& \\ P(x\_1, x\_2, ..., x\_n)]$$

A sequence comprehension defines a sequence whose elements are derived from the evaluation of expression $e(x\_1, x\_2, ..., x\_n)$ under the condition that x_1 takes values from type T_1, x_2 from T_2, ..., x_n from T_n, and all of these values satisfy property $P(x\_1, x\_2, ..., x\_n)$. If unnecessary, the bindings can be omitted. Thus we may use another form of sequence comprehension:

$$[e(x\_1, x\_2, ..., x\_n) \mid P(x\_1, x\_2, ..., x\_n)]$$

Note that all the types T_i (i=1..n) are countable numeric types and the elements of the sequence must occur in an *ascending* order. For example,

$$[i * j \mid i: \mathbf{nat}, j: \mathbf{nat} \ \& \ 1 <= i + j <= 3] = [1, 2, 2]$$

As with the set notation, we also use the following special notation to represent a sequence of integer interval from i to j:

$$[i, ..., j] = [x \mid x: \mathbf{int} \ \& \ i <= x <= j]$$

Thus,

$$[3, ..., 6] = [3, 4, 5, 6]$$
$$[-2, ..., 2] = [-2, -1, 0, 1, 2]$$
$$[0, ..., 4] = [0, 1, 2, 3, 4]$$

However, if index j is smaller than index i, $[i, ..., j]$ represents the empty sequence [ ]. For example,

$$[9, ..., 2] = [\ ]$$

9.3.2 Operators

Sequences can be manipulated by sequence operators. Some operators take a sequence and yield a value related to its elements, while other operators take several sequences and yield another sequence.

In the discussions of sequence operators below, we assume that T is the element type for building up sequences. Each operator is explained by giving both a formal definition and examples to help in the understanding of its meaning.

Length

The length of a sequence means the number of its elements. The length operator is denoted by symbol **len**, and is defined as

> **len**: **seq of** T $->$ **nat0**
> **len**(s) $==$ the number of elements in s

For example, let s1 $=$ [4, 9, 10], s2 $=$ [{3, 9}, {6}], s3 $=$ [10, 9, 4, 25], and s4 $=$ "university". Then,

> **len**(s1) $= 3$
> **len**(s2) $= 2$
> **len**(s3) $= 4$
> **len**(s4) $= 10$

Note that s2 is a sequence of set values, so its length is the number of all the set values occurring in the sequence.

Sequence application

A sequence can apply to an index, a natural number, to yield the element occurring at the position indicated by the index. Let s be a sequence of type **seq of** T. Then, s can be regarded as a function from **nat** to T:

> s: **nat** $->$ T
> s(i) $==$ the ith element of sequence s

The precondition for applying s to an index i is that index i is within the range of 1 to **len**(s). Otherwise, if i is beyond this range, the sequence application s(i) is undefined. For example,

> s1(1) $= 4$
> s1(2) $= 9$
> s2(1) $=$ {3, 9}

s3(4) = 25
s4(5) = 'e'

Subsequence

A subsequence of a sequence is part of the sequence. Let **s** be a sequence of type **seq of** T, and i and j be two indexes. Then, the subsequence of **s** that keeps the elements in the same order as they are in **s** is denoted as

s(i, j): **nat * nat** –> **seq of** T
s(i, j) == [s(i), s(i + 1), ..., s(j - 1), s(j)]

Thus,

s1(2, 3) = [9, 10]
s1(1, 3) = s1
s3(2, 4) = [9, 4, 25]
s4(2, 8) = "niversi"

Head

The head of a non-empty sequence is its first element. The head operator is denoted by symbol **hd**, and is defined as

hd: **seq of** T –> T
hd(s) == if len(s) > 0
        then s(1)
        else nil

For example,

hd(s1) = 4
hd(s2) = {3, 9}
hd(s3) = 10
hd(s4) = 'u'

It is not difficult to understand the head elements of **s1** and **s3**, but one may be a little puzzled when looking at the result of **hd(s2)**, because the result is not a simple number but a set of numbers. In fact, since a sequence can contain any type of value, the result of an application of the operator can be either a value of a basic type or a value of a compound type, as long as the type is the element type of the sequence. Note that if **s** is the empty sequence, **hd(s)** is undefined.

Tail

The tail of a non-empty sequence is its subsequence resulting from eliminating its head. The tail operator is denoted by the symbol **tl**, and is defined as

tl: **seq of** T –> **seq of** T
tl(s) == s(2, **len**(s))

The application of the operator **tl** to the empty sequence is undefined, that is, **tl**([ ]) = **nil**. For example,

tl(s1) = [9, 10]
tl(s2) = [{6}]
tl(s3) = [9, 4, 25]
tl(s4) = "niversity"

Elements

The operator for obtaining the set of all the elements of a sequence is **elems**:

elems: **seq of** T –> **set of** T
elems(s) == {x | x: T & (**exists**[i: {1, ..., **len**(s)}] | x = s(i)}

Since the result of **elems**(s) is a set, not a sequence, duplication of elements is not allowed in it. Thus,

elems(s1) = {4, 9, 10}
elems(s2) = {{3, 9}, {6}}
elems(s3) = {10, 9, 4, 25}
elems([5, 10, 5, 10, 15]) = {5, 10, 15}
elems(s4) = {'u', 'n', 'i', 'v', 'e', 'r', 's', 'i', 't', 'y'}

If s is the empty sequence, **elems**(s) is the empty set, that is,

elems([ ]) = { }.

Indexes

A sequence corresponds to a set of natural numbers that indicates the positions of the elements in the sequence. Such a set is known as *index set*. The operator for obtaining an index set of a sequence is **inds**:

inds: **seq of** T –> **set of** **nat**
inds(s) == {i | i: **nat** & **exists**[x: **elems**(s)] | s(i) = x}

It is obvious that the index set of the empty sequence is the empty set. Furthermore, the cardinality of **inds**(s) is equal to the length of sequence s, but

may be greater than that of the element set **elems(s)** due to the possibility of duplicate elements in **s**. For example, suppose s1 = [4, 9, 10], s2 = [{3, 9}, {6}], s3 = [10, 9, 4, 25], and s4 = "university". Then,

> **inds**(s1) = {1, 2, 3}
> **inds**(s2) = {1, 2}
> **inds**(s3) = {1, 2, 3, 4}
> **inds**(s4) = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

The index set is often used when describing a property of a sequence. Consider the example

> **exists**[i: **inds**(s)] | s(i) > 5

This quantified expression describes the property of sequence **s**: that **s** has at least one element greater than 5.

Concatenation

Sequences can be concatenated to form another sequence. The operator for sequence concatenation is **conc**, and is defined in an implicit manner as

> **conc(s_1: seq of T, s_2: seq of T) cs: seq of T**
> **post len(cs)** = **len**(s_1) + **len**(s_2) **and**
>   (**forall**[i: **inds**(s_1)] | cs(i) = s_1(i)) **and**
>   (**forall**[i: **inds**(s_2)] | cs(i + **len**(s_1)) = s_2(i))

The concatenation of sequences s_1 and s_2 is formed by appending s_2 to the end of s_1. For example,

> **conc**(s1, s3) = [4, 9, 10, 10, 9, 4, 25]
> **conc**(s4, s4) = "universityuniversity"

The concatenation of sequences is not commutative. Thus,

> **conc**(s1, s3) <> **conc**(s3, s1)

because

> **conc**(s3, s1) = [10, 9, 4, 25, 4, 9, 10]

which is different from **conc**(s1, s3) given above.
    The concatenation operator **conc** can be extended to deal with more than two sequences. Thus,

$$\textbf{conc}(s\_1, s\_2, ..., s\_n) = \textbf{conc}(s\_1, \textbf{conc}(s\_2, \textbf{conc}(s\_3, ...)))$$

For example, let s1 = [5, 15, 25], s2 = [10, 20, 30, 40], and s3 = [2, 4, 6, 8, 10]. Then,

conc(s1, s2, s3) = [5, 15, 25, 10, 20, 30, 40, 2, 4, 6, 8, 10]

Distributed concatenation

As we have mentioned before, the elements of a sequence can be values of any type available in SOFL, so it is possible to have sequences whose elements are again sequences. Such a sequence is called *sequence of sequences*. Let S be a sequence of sequences:

S = [s_1, s_2, ..., s_n]

where s_i (i=1..n) are sequence values of a sequence type. Then, the operator for the distributed concatenation of S is defined as

dconc: **seq of seq of** T **-> seq of** T
dconc(S) == **conc**(s_1, s_2, ..., s_n)

For example, let S1 = [[5, 15, 25], [10, 20, 30, 40], [2, 4, 6, 8, 10]] and S2 = [[{2, 3, 4}, {7}, {8, 9}], [{10, 20}], [{50, 100, 150}, {30, 60}]]. Then,

dconc(S1) = [5, 15, 25, 10, 20, 30, 40, 2, 4, 6, 8, 10]
dconc(S2) = [{2, 3, 4}, {7}, {8, 9}, {10, 20}, {50, 100, 150}, {30, 60}]

Since each element of a string value can only be a character, not another sequence, the distributed concatenation operator **dconc** cannot be applied to string values.

Equality and inequality

Sequences can be compared to determine whether they are identical or not. As in many examples of this chapter, the operators for equality and inequality are = and <>, respectively:

s_1 = s_2 <=>
len(s_1) = **len**(s_2) **and forall**[i: **inds**(s_1)] | s_1(i) = s_2(i)

s_1 <> s_2 <=>
not s_1 = s_2

For example, let s1 = [5, 15, 25], s2 = [10, 20, 30, 40], and s3 = [2, 4, 6, 8, 10]. Then,

s1 = s1 <=> **true**
s1 <> s2 <=> **true**
s2 = s3 <=> **false**

9.4 Specifications Using Sequences

In this section, let us look at two examples of specifications with the sequence notation. The first example describes an *input and output* module that takes care of inputting data from an input device (e.g., keyboard, file) and outputting data to an output device (e.g., display, file). The second example models a simplified *membership management system* of a club using a module called MembershipManagementSystem. This system is intended to deal with the registration of members, searching of members, and exchange of members in the member list.

9.4.1 Input and Output Module

The input and output module is named InputOutput. To model the input and output processes, we first need to model the input device and output device using appropriate data types. Since the order of elements in an input and output device is significant, and there is the possibility of having duplicate elements on such a device, we model them as sequences of characters. On the basis of this data modeling, the three processes are provided: Input, Output, and Delete. The process Input reads a character from the input device whereas Output outputs a character to the output device. The process Delete removes the last character from the current output device. The formal specification of this module is as follows:

> **module** InputOutput;
>
> **var**
>
> input_device, output_device: **seq of char**;
>
> **process** Input() ic: **char**
> **ext wr** input_device
> **pre**   input_device $<> [\,]$
> **post conc**([ic], input_device) = ˜input_device
> **comment**

The precondition of this process requires that input_device not be empty. The postcondition describes that the head of input_device before the process is read and bound to the output variable ic, and that input_device is updated by removing its head character.

> **end_process**;
>
> **process** Output(oc: **char**)
> **ext wr** output_device
> **post** output_device = **conc**(˜output_device, [oc])

comment

The output of character oc to output_device is defined by a sequence concatenation of the initial output_device and the sequence composed of element oc.

end_process;

process Delete() dc: **char**
ext wr output_device
pre   output_device $<>$ [ ]
post ~output_device $=$ **conc**(output_device, [dc])
comment

This process removes the last character of output_device, which is reflected by defining the initial output_device as a concatenation of the final output_device and the sequence composed of character dc.

end_process;
end_module;

9.4.2 Membership Management System

The first step in modeling this system is to define a data structure recording all the members in the club. Since the order of joining the club may affect the right of the members in the club, and it is possible to have members who share the same name, we model the list of members as a sequence. In addition, three processes are provided: Register, Search, and Exchange. The process Register records a new member in the member list; Search provides a set of indexes of a requested member in the member list; and Exchange makes two members exchange their positions in the member list. The module is formally specified as follows:

module MembershipManagementSystem;

type

Member $=$ **string**; /* A member is denoted by its name which is a string of characters */

var

all_members: **seq of** Member;

process Register(m: Member)
ext wr all_members
post all_members $=$ **conc**(~all_members, [m])

comment

The function for recording member m in the member list all_members is specified by defining the final all_members as a concatenation of the initial all_members and the sequence composed of member m.

  end_process;

  process Search(m: Member) pos: **set of nat**
  ext rd all_members
  post pos = {i | i: **nat** & all_members(i) = m}
  comment
  Finding all the positions of member m in the member list all_members is modeled by a set comprehension.
  end_process;

  process Exchange(pos1, pos2: **nat**)
  ext wr all_members
  pre pos1 inset inds(all_members) **and** pos2 inset inds(all_members)
  post all_members(pos1) = ~all_members(pos2) **and**
        all_members(pos2) = ~all_members(pos1) **and**
        forall[i: inds(all_members)] | i <> pos1 **and** i <> pos2 =>
                          all_members(i) = ~all_members(i)
  comment
This process exchanges only the members at position pos1 and pos2, and keeps the rest of the members unchanged in the list

  end_process;
  end_module.

9.5 Exercises

1. Given a set T = {1, 2, 5}, define a sequence type based on T, and give ten possible sequence values in the type.
2. Evaluate the following sequence comprehensions:
    a) [x | x: **nat** & 3 < x < 8]
    b) [y | y: **nat0** & y <= 3]
    c) [x - y | x: **nat0**, y: **nat0** & 1 < x + y < 3]
3. Let s1 = [5, 15, 25], s2 = [15, 30, 50], s3 = [30, 2, 8], and s = [s1, s2, s3]. Evaluate the following expressions:
    a) **hd(s1)**
    b) **hd(s)**
    c) **len(tl(s1)) + len(tl(s2)) + len(tl(s3))**
    d) **len(s1) + len(s2) - len(s3)**
    e) **union(elems(s1), elems(s2))**

   f) **inter(union({hd(s2)}, elems(s3)), elems(s1))**
   g) **union(inds(s1), inds(s2), inds(s3))**
   h) **elems(conc(s1, s2, s3))**
   i) **dconc(s)**

4. Construct a module to model a queue of integers with the processes Append, Eliminate, Read, and Count. The process Append adds a new element to the queue; Eliminate deletes the top element of the queue; Read returns the top element; and Count yields the number of the elements in the queue.

5. Write a specification for a process Search. The process takes an integer and searches through a sequence of integers. If the input integer is found in the sequence, its indexes (there might be more than one occurrences of the input integer in the sequence) are given as the result. If the input integer is not found, then the empty set is given as the output. Note that the sequence of integers must be treated as an external variable of the process.

10

The Composite and Product Types

It is often the case that an object in the real world has many attributes, each describing an aspect of the object. For example, a bank account is often associated with the attributes account name, account number, password, and balance; a student may be described by a name, identification number, age, department, and so on. The *composite types* introduced in this chapter provide a data structure for modeling such objects.

Sometimes, we may need to represent an object by several data items as a group in a certain order. For example, a date is characterized by year, month, and day, that is, a specific day of a year can be described by the three data items, year, month, and day, in a certain order. The representation of the date varies depending on countries. For example, Americans adopt the order month, day, year, whereas the British use the order day, month, year. Chinese and Japanese express a date in the order of year, month, day. The common feature of these three different representations is that the order of the occurrence of several data items in a group is important: changing the order may change the value of the data item group. The objects with such characteristics can be modeled by *product types* discussed in this chapter.

Composite types and product types share the similarity that a value of both types is composed of several data items. But they are different in the sense that the data items contained in a value of a composite type is referred to by name, so the order of the occurrences of the data items is not important, whereas a value of a product type is sensitive to the order of its data items.

In this chapter we first introduce composite types and then discuss product types.

10.1 Composite Types

10.1.1 Constructing a Composite Type

A composite type is constructed using the type constructor: **composed of ... end**. The general format of a composite type is:

composed of
f_1: T_1
f_2: T_2
   ...
f_n: T_n
end

where f_i (i=1..n) are variables called *fields* and T_i are their types. Each field is intended to represent an attribute of a composite object of the type. We can give a name A for this type in the form

A = **composed of**
      f_1: T_1
      f_2: T_2
      ...
      f_n: T_n
      end

A value of a composite type is called *composite object* or *composite value.* If variable co is declared as

   co: A;

or

   co: **composed of**
      f_1: T_1
      f_2: T_2
      ...
      f_n: T_n
      end

then the variable co can hold any values of type A. For example, in Chapter 4 we declare Account as a composite type of three fields:

   Account = **composed of**
               account_no: **nat1**
               password: **nat1**
               balance: **real**
               end

With this type, the variable account is declared as

   account: Account;

Note that it is not the fields of a composite type like Account that are associated with values, but the fields of a composite object like account that are associated with values.

10.1.2 Fields Inheritance

Fields inheritance provides a convenient mechanism for defining a new composite type based on an already defined one. Suppose we want to declare a composite type Q containing fields already defined in another existing composite type W; then, we adopt the following form for the type declaration:

> Q / W = **composed of**
>   b_1: Tb_1
>   b_2: Tb_2
>   ...
>   b_m: Tb_m
> **end**

Where type W is assumed to have been defined before in the form

> W = **composed of**
>   a_1: Ta_1
>   a_2: Ta_2
>   ...
>   a_n: Ta_n
> **end**

Thus, type Q is defined as a composite type that contains all the fields a_1, a_2, ..., a_n of type W, and b_1, b_2, ..., b_m are defined explicitly in type Q. In this case, we say type Q inherits from type W. However, this kind of inheritance is syntactical inheritance and there is a strict constraint on the order of the inherited fields (e.g., a_1, ..., a_n) in the current type (e.g., Q): the inherited fields are all assumed to be declared before the fields declared explicitly in the current type (e.g., b_1, ..., b_m). Applying this rule, the declaration of type Q above is in fact equivalent to the following declaration:

> Q = **composed of**
>   a_1: Ta_1
>   a_2: Ta_2
>   ...
>   a_n: Ta_n
>   b_1: Tb_1
>   b_2: Tb_2
>   ...
>   b_m: Tb_m
> **end**

To ensure simplicity and avoid possible confusion in field names, we allow a composite type to inherit from *only one* other composite type. Thus, the following declaration is not allowed:

Q / W, A = **composed of**

...

      end

10.1.3 Constructor

There is only one constructor that is used to generate composite values of composite types. This constructor is known as *make-function*, and its general format is

mk_A(v_1, v_2, ..., v_n)

The make-function yields a composite value of composite type A whose field values are v_i (i=1..n) that corresponds to fields f_1, f_2, ..., f_n, respectively. For example,

mk_Account(1073548, 1234, 5000)

makes a composite value of type Account whose account_no is 1073548, password is 1234, and balance is 5000. If we write

account = **mk_Account(1073548, 1234, 5000)**

then the account_no of the variable account is 1073548, password is 1234, and balance is 5000.

10.1.4 Operators

Two kinds of operators are available to deal with composite values. One is called *field select* and the other is called *field modification*.

Field select

Let co be a variable of composite type A, as defined in Section 10.1.1. Then, we use

co.f_i

to represent field f_i (i=1..n) of composite object co. For example,

account.password

refers to the field password of composite value account, and

```
account.balance
```

refers to the field balance of account.

Field modification

Given a composite value, say co, of type A, we can apply the field modification operator **modify** to create another composite value of the same type, but with possibly different field values. The format of the operator is

$$\textbf{modify}(\text{co}, \text{f\_1} \to \text{v\_1}, \text{f\_2} \to \text{v\_2}, ..., \text{f\_n} \to \text{v\_n})$$

The result of this application of the operator **modify** to composite value co and the pairs f_1 -> v_1, f_2 -> v_2, ..., f_n -> v_n is a composite value whose values of fields f_1, f_2, ..., f_n are v_1, v_2, ..., v_n, respectively. Let us take composite value account of type Account, given previously as an example. As defined before,

```
account = mk_Account(1073548, 1234, 5000)
```

The application of **modify** to account and field password yields value

```
account1 = modify(account, password -> 4321)
```

Thus, the password field of account1 is 4321 and the other two fields remain the same as those of account. That is,

```
account1 = mk_Account(1073548, 4321, 5000)
```

Applying the modify operator again to account1, we generate another value account2:

```
account2 = modify(account1, balance -> 10000)
```

This is equivalent to:

```
account2 = mk_Account(1073548, 4321, 10000)
```

Note that the operator **modify** does not change the current composite value. For example, after the evaluation of **modify**(account, password -> 4321), account remains the same as before the evaluation, that is, account = **mk_Account**(1073548, 1234, 5000) still holds.

10.1.5 Comparison

Two composite values can be compared to determine whether they are identical or not. Suppose co1 and co2 are two composite values of type A. Then,

co1 = co2

means that co1 and co2 have the same type and all their field values are exactly the same, respectively. Thus,

mk_Account(1073548, 1234, 5000) = **mk**_Account(1073548, 1234, 5000)
mk_Account(1073548, 4321, 5000) =
        modify(**mk**_Account(1073548, 1234, 5000), password −> 4321)

If the above condition is not met, the two composite values are not identical. For example,

mk_Account(1073548, 1234, 5000) <> **mk**_Account(1073548, 4321, 5000)
mk_Account(1073548, 4321, 5000) <> **mk**_Account(1073548, 4321, 10000)

10.2 Product Types

A product type defines a set of *tuples* with a fixed length. A tuple is composed of a list of values of possibly different types. Let $T\_1, T\_2, ..., T\_n$ be n types. Then, a product type T is defined as follows:

T = T_1 * T_2 * ... * T_n

A value of T is expressed as

mk_T(v_1, v_2, ..., v_n)

where v_1 **inset** T_1, v_2 **inset** T_2, ..., v_n **inset** T_n. The values v_i (i=1..n) are called *elements* of this tuple. For example, suppose type Date is declared as

Date = **nat0** * **nat0** * **nat0**

Then, the tuples

mk_Date(1999, 7, 25)
mk_Date(2000, 8, 30)
mk_Date(2001, 7, 10)

are all values of type Date, where **mk**_Date is a make-function for product type Date. If a variable d is declared with type Date as

d: Date

then, we can use the following expressions in specifications:

d = **mk**_Date(1999, 7, 25)
d = **mk**_Date(2000, 8, 30)
d = **mk**_Date(2001, 7, 10)

There are two operations on tuples: *tuple application* and *tuple modification*. A *tuple application* yields an element of the given position in the tuple, whose general format is

a(i)

where a is a variable of product type; and i is a natural number indicating the position of the element referred to in tuple a. The result of a(i) is the ith value in the tuple a. For example, let

date1 = **mk**_Date(1999, 7, 25)
date2 = **mk**_Date(2000, 8, 30)

Then, the following results can be derived:

date1(1) = 1999
date1(2) = 7
date1(3) = 25
date2(1) = 2000
date2(2) = 8
date2(3) = 30

Or tuples can be directly used in applications, such as

mk_Date(2000, 8, 30)(2) = 8
mk_Date(2000, 8, 30)(3) = 30

A tuple modification is similar to a composite value modification. The same operator **modify** is also used for tuple modification, but with slightly different syntax:

modify(tv, 1 –> v_1, 2 –> v_2, ..., n –> v_n)

This operation yields a tuple of the same type based on the given tuple tv, with the first element being v_1, the second element being v_2, and so on. Unlike the application of operator **modify** to composite objects, the indexes of the elements of a tuple, rather than the field names, are given in the argument list. The signature of the **modify** is

Table 10.1. A student record table

personal data	course1	course2	total
Helen, 0001, A3	2	2	4
Alexis, 0002, A2	0	2	2
...	...	...	...

modify: T * (nat * T_1) * (nat * T_2) * ... * (nat * T_n) -> T

where T is a product type, and T_i (i=1..n, n >= 1) are the element types. For example,

modify(mk_Date(2000, 8, 30), 1 -> 2001, 3 -> 20) = (2001, 8, 20)
modify(mk_Date(2001, 8, 20), 2 -> 15) = (2001, 15, 20)

As with composite values, tuples can also be compared with each other to determine whether they are the same or not. Suppose t1 and t2 are two tuples, then t1 = t2 means that t1 and t2 are the values of the same type and they have exactly the same elements in the same order. However, if this condition is not satisfied, the two tuples are not identical, that is, t1 <> t2.

Using values of product types together with set values or sequence values, we can build *relations* and *tables*. A relation is a set of pairs that describes an association between members of its domain and those of its range, but with no restriction on the type of association. Thus, one element in the domain may be associated with many members in the range, and vice versa. A table is a sequence of tuples, such as the truth tables for logical operators **and**, **or**, and **not** given in Chapter 2. In the next section we give an example to explain how composite and product types are used for data abstraction in specifications.

10.3 An Example of Specification

Suppose we want to build a table to record students' credits resulting from two courses, like Table 10.1. Each row of the table corresponds to one student and has four columns presenting personal data, the credits of the two courses, and the total credit. The personal data includes name, identification number, and class to which the student belongs, which are denoted by the fields name, id, and class, respectively, in the table. Many processes can be built to manipulate data of the table, but to keep the description as brief and comprehensible as possible, we only give the two processes Search and Update, which are specified in the module Students_Record.

module Students_Record;

type

CourseCredit = **nat0**;

TotalCredit = **nat0**;

PersonalData = **composed of**
                        name: **string**
                        id: **nat0**
                        class: **string**
                        end;

OneStudent = PersonalData * CourseCredit * CourseCredit * TotalCredit;

StudentsTable = **seq of** OneStudent;

var

 students_table: StudentsTable;

inv

forall[i, j: **inds**(students_table)] |
        i <> i => students_table(i)(1).id <> students_table(j)(1).id;

process Search(search_id: **nat0**) info: OneStudent
ext rd students_table
pre exists[i: **inds**(students_table)] | students_table(i)(1).id = search_id
post exists![i: **inds**(students_table)] | students_table(i)(1).id = search_id
                                and
                                info = students_table(i)]
end_process;

process Update(one_student: OneStudent, credit1, credit2: CourseCredit)
ext wr students_table
pre   exists[i: **inds**(students_table)] | students_table(i) = one_student
post   len(students_table) = **len**(~students_table) **and**
        forall[i: **inds**(~students_table)] |
                (~students_table(i) = one_student =>
                  students_table(i) =
                    modify(~students_table(i), 2 -> credit1, 3 -> credit2,
                            4 -> credit1 + credit2)) **and**
                (~students_table(i) <> one_student =>
                  students_table(i) = ~students_table(i))
end_process;

end_module;

In this module, several types including CourseCredit, TotalCredit, and PersonalData are declared in order to declare type OneStudent. The type OneStudent is declared as a product type composed of the three types: PersonalData, CourseCredit, and TotalCredit. Based on type OneStudent, the type StudentsTable is defined as a sequence type.

The state variable students_table is declared with the type StudentsTable. This variable must satisfy the invariant given in the **inv** section, which requires that no different elements representing different students' data share the same identification number. In this invariant, students_table(i), a sequence application to index i, denotes a tuple of the product type OneStudent, and students_table(i)(1) is a tuple application representing the first element of the tuple (i.e., personal data). This first element is a value of composite type PersonalData, so students_table(i)(1).id denotes the identification number field of the composite value.

The process Search provides the entire data of the student whose identification number is the same as search_id, provided as the input. To ensure that this process behaves correctly, the student to be searched must exist in students_table. Since there are no elements in the table that share the same identification number according to the invariant, we use the quantifier **exists!**, rather than **exists**, in the postcondition.

The function of process Update is to modify the credits of the two courses of student one_student with the given credits credit1 and credit2. To ensure that the modification can be conducted, the given student one_student must exist in the table. This requirement is specified as the precondition of the process. The postcondition is given as a universally quantified expression that describes the relation between the initial value and the final value of the state variable students_table by modifying the corresponding attributes of the student having the same identification number as that of one_student with the given credits credit1 and credit2. The total credit of the student to be updated is derived by adding up credit1 and credit2. Except for this change, all the other students' data in the sequence students_table remain unchanged.

10.4 Exercises

1. Explain the similarity and difference between a composite type and a product type.
2. Let a = **mk_Account**(010, 300, 5000), where the type Account is defined in Section 10.1.1. Evaluate the following expressions:
   a) a.account_no
   b) a.password
   c) a.balance
   d) **modify**(a, password -> 250)
   e) **modify**(**mk_Account**(020, 350, 4050), account_no -> 100, balance -> 6000)

3. Let x be a variable of the type **Date** defined in Section 10.2, and x = **mk_Date**(2002, 2, 6). Evaluate the following expressions:
   a) x(1)
   b) x(2)
   c) x(3)
   d) **modify**(x, 1 –> 2003)
   e) **modify**(x, 2 –> 5, 3 –> 29)
   f) **modify**(x, 1 –> x(1), 2 –> x(2))

4. Define a composite type **Student** that has the fields **name**, **data_of_birth**, **college**, and **grade**. Write the specifications for the processes **Register**, **Change_Name**, and **Get_Info**. **Register** takes a value of **Student** and adds it to an external variable **student_list**, which is a sequence of students. **Change_Name** updates the name of a given student with a given name. **Get_Info** provides all the available field values for a given student name (assuming that the student name is unique).

11

The Map Types

Different level associations between two sets can be defined by different mathematical notions. A relation defines a relaxed association by providing a set of pairs: an element in the domain may be associated with several elements in the range, and vice versa. A function defines a more restricted association between elements in its domain and range: an element in the domain can be associated with only one element in the range. Such an association is often known as *mapping*. As we have introduced in Chapter 4, the domain and range of a function can be infinite sets (e.g., **nat0**, **int**, **real**), therefore the function itself can describe an infinite mapping from its domain to its range. However, it is often sufficient to use finite mappings in software specifications because computer software deals with a finite number of data items. For example, a table describing cars and their makers usually contains a finite number of pairs with the property that a car is made by only one maker.

In this chapter, we introduce map types by discussing the important issues, such as the definition of map types, the manipulation of map values, and the application of map types in process specifications.

11.1 What Is a Map

A map is a *finite* set of pairs, describing a mapping between two sets. The set whose elements are to be mapped to another set is known as the *domain* of the map, while the set whose elements are to be mapped to from the domain is called the *range* of the map. Figure 11.1 illustrates a simple map whose domain is the set {a, b, c, d} and range is {1, 2, 3}. Both a and b in the domain are mapped to 2 in the range; c is mapped to 3; and d is mapped to 1.

A map (or map value) is represented with a notation similar to the set notation:

{a_1 –> b_1, a_2 –> b_2, ..., a_n –> b_n}
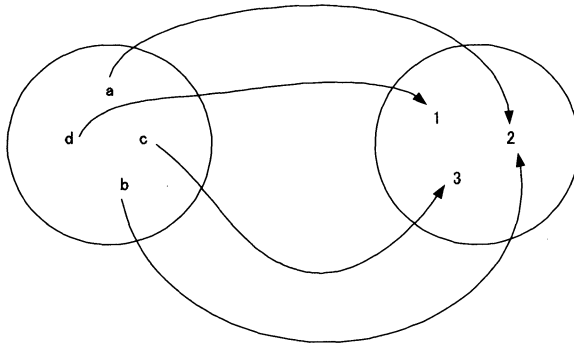
Fig. 11.1. A simple map

Each $a\_i \to b\_i$ (i=1..n) denotes a pair which is known as *maplet*, indicating that a_i in the domain is mapped to b_i in the range. For example, the map illustrated in Figure 11.1 is given as follows:

{a –> 2, b –> 2, c –> 3, d –> 1}

Since a map is a set of maplets, the order of maplets are not significant, that is, changing the order of maplets of a map does not change the map itself. As mentioned in the beginning of this chapter, a map describes a *many-to-one* mapping in general: it allows the mapping from many elements in the domain to the same element in the range, but does not allow the mapping from the same element in the domain to different elements in the range. The empty map is given in the form

{–>}

It has the same meaning as the empty set, but takes a different syntax from that of the empty set in order to help type checking of specifications.

11.2 The Type Constructor

The map type constructor is a function that constructs a map type for the given domain type T1 and range type T2, and the constructed type is written as

map T1 to T2

The map type provides a maximum mapping from T1 to T2, and any map value of this type defines a subset of this maximum mapping. Note that T1

and T2 in the map type may be any kinds of types, including infinite types, such as **nat** and **int**, but all map values of the map type must be finite. For example, given the type:

   map nat to char

the following map values can be constructed:

   {1 –> 'a', 2 –> 'b', 3 –> 'c', 4 –> 'd'}
   {5 –> 'u', 15 –> 'v', 25 –> 'w'}
   {10 –> 'x', 20 –> 'y'}
   {50 –> 'r'}
   {–>}

As with the other types introduced so far, we can also declare map type identifiers and then use them to declare variables in modules.

11.3 Operators

In this section, we explain all the operators defined on the map types. The operators include two kinds: *constructors* and *operators*. The constructors provide a way to form a map value from the element types, whereas the other operators manipulate map values.

11.3.1 Constructors

There are two constructors: *map enumeration* and *map comprehension*. A map enumeration is given as a set of maplets, as we have illustrated in Section 11.1. Its general format is:

   {a_1 –> b_1, a_2 –> b_2, ..., a_n –> b_n}

For example, the following are some maps:

   {3 –> 'a', 8 –> 'b', 10 –> 'c'}

   {"Hosei University" –> "Japan",
   "University of Manchester" –> "U.K.",
   "Jiaotong University" –> "China"}

   {1 –> s(1), 2 –> s(2), 3 –> s(3)}
where s is a sequence of integers containing three elements.
   A map comprehension is similar to a set comprehension, except that the elements of such a set are maplets. The map comprehension takes the form

{a −> b | a: T1, b: T2 & P(a, b)}

This expression defines a map composed of all those maplets whose elements a and b satisfy the property P(a, b). As with set types, if they are unnecessary or obvious, the bindings in the map comprehension can be omitted. Thus, the following form of map comprehension can be possible:

{a −> b | P(a, b)}

It is also essential that such a map comprehension does not violate the fundamental invariant of map values that one element in the domain can be mapped to only one element in the range. For example,

{x −> y | x: {5, 10, 15}, y: {10, 20, 30} & y = 2 * x} =
{5 −> 10, 10 −> 20, 15 −> 30}

defines a legal map, but the following map comprehension defines an illegal map:

{x −> y | x: {1, 2, 3}, y **inset** {5, 10, 15, 20} & y > x * 5} =
{1 −> 10, 1 −> 15, 1 −> 20, 2 −> 15, 3 −> 20}

11.3.2 Operators

The operators on map types take some maps as their arguments and yield other maps. All the operators available on map types are discussed below one by one.

Map application

Let m be a map:

m: **map** T1 **to** T2;

Then, m can be applied to an element in its domain to yield an element in its range. The map application takes the same syntax as that of the function application. Thus,

m(a)

denotes an application of map m to element a. For example, let

m1 = {5 −> 10, 10 −> 20, 15 −> 30}

Then,

m1(5) = 10
m1(10) = 20
m1(15) = 30

Domain and range

Let m be a map:

    m: **map** T1 **to** T2;

Then the domain of m is a subset of T1 and its range is a subset of T2, which can be obtained by applying the operators **dom** and **rng**, respectively. When applied to a map, the domain operator **dom** yields the set of the first elements of all the maplets in the map:

    dom: **map** T1 **to** T2 –> **set of** T1
    dom(m) == {a | a: T1 & **exists**[b: T2] | m(a) = b}

For example, by applying **dom** to the map m1 given previously, we derive

    dom(m1) = {5, 10, 15}

The range operator **rng**, when applied to a map, yields the set of the second elements of all the maplets in the map. This operator is formally defined as follows:

    rng: **map** T1 **to** T2 –> **set of** T2

    rng(m) == {m(a) | a **inset dom**(m)}

Applying **rng** to the map m1, we get

    rng(m1) = {10, 20, 30}

Domain and range restriction to

Given a map and a set, we may sometimes want to obtain the submap of the map whose domain or range is restricted to the set. Such operations are known as *domain restriction to* and *range restriction to*, respectively. The operator for domain restriction to is **domrt** and the operator for range restriction to is **rngrt**:

    domrt: **set of** T1 * **map** T1 **to** T2 –> **map** T1 **to** T2
    domrt(s, m) == {a –> b | a **inset inter**(s, **dom**(m)) **and** b = m(a)}
    rngrt: **map** T1 **to** T2 * **set of** T2 –> **map** T1 **to** T2

rngrt(m, s) == {a -> b | a: **dom**(m) & b = m(a) **and** b **inset inter**(s, **rng**(m))}

For example, let m1 = {5 -> 10, 10 -> 20, 15 -> 30} and s1 = {5, 10}. Then,

domrt(s1, m1) = {5 -> 10, 10 -> 20}
rngrt(m1, s1) = {5 -> 10}

Domain and range restriction by

In contrast with "domain restriction to" and "range restriction to" operations, we may sometimes want to derive a submap of a map whose domain or range is the subset of the domain or range of the map that is disjoint with a given set. Such operations are known as *domain restriction by* and *range restriction by*, respectively. The operators for these two operations are **domrb** and **rngrb**, respectively, and they are defined as follows:

domrb: **set of T1 * map T1 to T2 -> map T1 to T2**
domrb(s, m) == {a -> b | a **inset diff**(**dom**(m), s) **and** b = m(a)}

rngrb: **map T1 to T2 * set of T2 -> map T1 to T2**
rngrb(m, s) == {a -> b | a: **dom**(m) & b = m(a) **and** b **inset diff**(**rng**(m), s)}

domrb(s, m) yields a submap of map m whose domain is the subset of **dom**(m) disjoint with set s, while **rngrb**(m, s) denotes the submap of m whose range is the subset of **rng**(m) disjoint with s.

For example, the following results are derived by applying these two operators to the map m1 and the set s1 given previously:

domrb(s1, m1) = {15 -> 30}
rngrb(m1, s1) = {10 -> 20, 15 -> 30}

In fact, the result of **domrb**(s1, m1) is obtained by eliminating the maplets from the map m1 whose first element is a member of set s1, while the result of **rngrb**(m1, s1) is obtained by removing the maplets whose second element belongs to set s1.

Override

Overriding is an operation of performing a union of two maps m1 and m2, denoted by **override**(m1, m2), with the restriction that if a maplet in map m2 shares the first element with a maplet in m1, the resulting map includes only the maplet in m2 as its element. Formally, the operator **override** is defined as follows:

override: **map T1 to T2 * map T1 to T2 -> map T1 to T2**
override(m1, m2) == {a -> b | a: **union(dom**(m1), **dom**(m2)) &
a **inset dom**(m2) => b = m2(a) **and**
a **notin dom**(m2) => b = m1(a)}

For example, let m1 = {5 -> 10, 10 -> 20, 15 -> 30}, m2 = {10 -> 5, 15 -> 50, 4 -> 20}. Then,

override(m1, m2) = {10 -> 5, 15 -> 50, 4 -> 20, 5 -> 10}

Such an operation can be done in several different ways, but a simple way to do it is to first include all the maplets in m2 in the resulting map **override**(m1, m2), and then expand it by including those maplets in m1 whose first elements are not in the domain of m2 (i.e., **dom**(m2)). Note that **override** is not commutative, that is, **override**(m1, m2) <> **override**(m2, m1) in general, as we can see by comparing **override**(m1, m2) given above and **override**(m2, m1) given here

override(m2, m1) = {5 -> 10, 10 -> 20, 15 -> 30, 4 -> 20}

However, if the domains of maps m1 and m2 are *disjoint*, the overriding operation is the same as the *union* of m1 and m2 (regarding these two maps as sets of maplets). Only under this restriction is the **override** operator commutative, that is,

override(m1, m2) = **override**(m2, m1)

Suppose we change the map m1 given previously to the map

m1 = {5 -> 10, 8 -> 20, 2 -> 30}

Then, the following result can be easily derived:

override(m1, m2) = **override**(m2, m1)

because

override(m1, m2) = **override**(m2, m1) =
{5 -> 10, 8 -> 20, 2 -> 30, 10 -> 5, 15 -> 50, 4 -> 20}

Map inverse

Map inverse is an operation that yields a map from a given map by exchanging the first and second elements of every maplet of the given map. The operator for the map inverse operation is **inverse**:

inverse: **map** T1 **to** T2 −> **map** T2 **to** T1
inverse(m) == {a −> b | a: **rng**(m), b: **dom**(m) & a = m(b)}

For example, by applying the **inverse** operator to map m1 = {5 −> 10, 8 −>
20, 2 −> 30} we get another map:

inverse(m1) = {10 −> 5, 20 −> 8, 30 −> 2}

However, we must bear in mind that if the map defines a *many-to-one* rather
than a *one-to-one* mapping between its domain and range, application of
the **inverse** operator is undefined, because it would yield a map violating
the fundamental property of maps that one element in the domain can be
mapped to only one element in the range. Consider the application of the
inverse operator to the map m2 = {10 −> 5, 15 −> 5, 4 −> 20} as an example:

inverse(m2) = {5 −> 10, 5 −> 15, 20 −> 4}

This result is no longer a map but a relation, because element 5 is mapped to
two different elements: 10 and 15.

Map composition

By composing two maps, we can construct a more complicated map. To make
such a composition possible, the range of the first map must share the same
type as for the domain of the second map. The operator for map composition
is **comp**:

comp: **map** T1 **to** T2 * **map** T2 **to** T3 −> **map** T1 **to** T3
comp(m1, m2) == {a −> b | a: **dom**(m1), b: **rng**(m2) &
               exists[x: **rng**(m1)] | x **inset** **dom**(m2) **and**
                  x = m1(a) **and** b = m2(x)}

For example, suppose m1 = {5 −> 10, 8 −> 20, 2 −> 4}, m2 = {10 −> 5, 15
−> 5, 4 −> 20}; then, the composition of m1 and m2 is:

comp(m1, m2) = {5 −> 5, 2 −> 20}

The composition is derived by considering the facts: m1(5) = 10 and m2(10)
= 5, m1(2) = 4, and m2(4) = 20, and there is no element 8 in the domain of
m2.

Like the **override** operator, **comp** is also not commutative. Thus, com-
position **comp**(m1, m2) is usually different from composition **comp**(m2, m1).
Sometimes, composition **comp**(m2, m1) may not make sense because of type
incompatibility (give a specific example as an exercise).

Equality and inequality

As with values of any other types, maps can also be compared to determine whether they are the same or not. We use m1 = m2 to mean m1 is identical to m2, and m1 <> m2 to mean m1 is different from m2. Formally,

m1 = m2 <=>
dom(m1) = **dom**(m2) **and rng**(m1) = **rng**(m2) **and**
forall[a: **dom**(m1), b: **rng**(m1)] | b = m1(a) <=> b = m2(a)
m1 <> m2 <=> **not** m1 = m2

The map m1 is identical to m2 if and only if they have exactly the same maplets. Otherwise, they are not identical.

Note that although a map is a set of maplets, the set operators are not applicable to maps because maps and sets are objects of different types. SOFL is a strong typed language that usually does not allow the operators on one type to be applied to the values of different types, except clearly polymorphic operators, such as +, =, <>, and **modify**.

11.4 Specification Using a Map

Let us reconsider the definition of type Account, declared in Section 4.15 of Chapter 4, with a map type. Since every customer's account number is unique and it is common to allow one customer to have only one account of the same kind in the same bank, the customer account can be modeled as a map from the account number to the account data, including password and balance. The real system dealing with customer accounts can be much more complex than the model we are discussing, but the primary purpose of our simplified example is to show how map types can be used to model data structures. The principle of this technique can be extended to deal with more complicated cases.

For brevity, we do not provide a full picture of the module in which the Account and the related processes are defined; rather, we give only the necessary parts in the module so that the problem can be well focused. First, the type Account is defined as a map type with the type AccountNumber being its domain and AccountData being its range:

Account = **map** AccountNumber **to** AccountData;
AccountNumber = **nat**;
AccountData = **composed of**
          password: **nat**
          balance: **real**
          end;

We then redefine the processes Check_Password, Withdraw, and Show_Balance by simplifying their interfaces pre and postconditions.

> **process** Check_Password(card_id: AccountNumber, pass: **nat**)
> confirm: **bool**
> **ext rd** account_file: Account
> **post** card_id **inset dom**(account_file) **and**
>   account_file(card_id).password = pass **and** confirm = true
>   **or**
>   card_id **notin dom**(account_file) **and** confirm = false
> **comment**

If the given account number card_id and password pass are matching with the account_file, the output confirm will be **true**; otherwise, it will be **false**.

> **end_process**;

> **process** Withdraw(card_id: AccountNumber, amount: **real**) cash: **real**
> **ext wr** account_file: Account
> **pre** card_id **inset dom**(account_file) and amount <=
>   account_file(card_id).balance
> **post** account_file = override(~account_file,
>   {card_id –> **mk_AccountData**(~account_file(card_id).password,
>   ~account_file(card_id).balance - amount)}) **and**
>   cash = amount
> **comment**

The precondition requires that the provided card_id be registered in the account_file and the requested amount to withdraw be smaller than or equal to the current balance. The updating of the current balance of the account with the account number card_id is expressed by a map overriding operation: the updated balance is the result of subtracting the requested amount from the current balance.

> **end_process**;

> **process** Show_Balance(card_id) bal: **real**
> **ext rd** account_file: Account
> **pre** card_id **inset dom**(account_file)
> **post** bal = account_file(card_id).balance
> **comment**

The account number card_id must exist in account_file before the execution of the process. The output variable bal is equal to the current balance, which is reflected by a map application in the postcondition.

> **end_process**;

The simplicity of process specifications may be affected by using different data structures for modeling the data items involved. The process specifications given in Section 4.15 of Chapter 4 are more complicated than those given

just above. Of course, in this particular case the complexity of the process specifications are due not only to the use of map data structure, but also due to the simplification of interfaces of the processes.

11.5 Exercises

1. Describe the similarity and difference between a map and a function.
2. Given two sets T1 = {1, 2}, T2 = {10, 11}, construct a map type with T1 being its domain type and T2 being its range type, and enumerate all the possible maplets of the map type.
3. Let m1 and m2 be two maps of the map type from **nat0** to **nat0**; m1 = {1 -> 10, 2 -> 3, 3 -> 30}, m2 = {2 -> 40, 3 -> 1, 4 -> 80}, and s = {1, 3}. Then, evaluate the following expressions:
   a) **dom**(m1)
   b) **dom**(m2)
   c) **rng**(m1)
   d) **rng**(m2)
   e) **domrt**(s, m1)
   f) **domrt**(s, m2)
   g) **rngrt**(m1, s)
   h) **rngrt**(m2, s)
   i) **domrb**(s, m1)
   j) **domrb**(s, m2)
   k) **rngrb**(m1, s)
   l) **rngrb**(m2, s)
   m) **override**(m1, m2)
   n) **override**(m2, m1)
   o) **inverse**(m1)
   p) **inverse**(m2)
   q) **comp**(m1, m2)
   r) **comp**(m2, m1)
   s) m1 = m2
   t) m1 <> m2
4. Give a concrete example to explain that **comp**(m1, m2) is defined, whereas **comp**(m2, m1) is undefined.
5. Define BirthdayBook as a map type from the type Person to the type Birthday, and specify the processes Register, Find, Delete, and Update. All the processes access or update the external variable birthday_book of the type BirthdayBook. The process Register adds a person's birthday to birthday_book; Find returns the birthday of a person in birthday_book; Delete eliminates the birthday for a person from birthday_book; and Update replaces the wrong birthday existing in birthday_book with a correct birthday.

12

The Union Types

A compound object may come from different types. For example, a component of a World Wide Web home page may contain normal text, pictures, audio data, and so on, each belonging to a different category. To model such compound objects using only one of the types introduced so far may not be sufficient. Types composed of several other types are needed to cope with this problem.

The union type is a solution to this problem. A union type is a union of several types: it contains all the elements of all the constituent types, and each element of the union type belongs to one of its constituent types.

In this chapter, we introduce the union types and the related operators. An example is given to illustrate the use of the union types.

12.1 Union Type Declaration

Let T1, T2, ..., Tn denote n types. Then, a union type T constituted of these types is declared in the format

$$T = T1 \mid T2 \mid ... \mid Tn$$

Thus, a value of T can come from one of the types T1, T2, ..., Tn. It is possible that types T1, T2, ..., Tn are not disjoint, but a union type should usually be formed by disjointed constituent types so that, for any value of type T, it can be precisely determined, by the **is** function (to be introduced later in this chapter), to which constituent type it belongs. For example, the union type Mixture that is composed of the three types **string**, **char** and **set of nat** is declared as

Mixture = **string** | **char** | **set of nat**

Thus, the following values belong to type Mixture:

    "Hosei University"
    "SOFL"
    'b'
    '5'
    {3, 5, 8}
    {10, 20, 100}

Since values of a union type are mixtures of values of different constituent types, it is difficult to build operators for manipulating them properly; but, as with the other types introduced so far, values of a union type can be compared to determine their equality and inequality. For example,

    "Hosei University" = "SOFL" <=> **false**
    'b' = 'b' <=> **true**
    '5' <> {3, 5, 8} <=> **true**
    "SOFL" = {10, 20, 100} <=> **false**

12.2 A Special Union Type

To specify some functions, it may need a union type that contains any possible value of any possible type in a specific system. For example, we may want a process to display the value of any possible type (onto an output device), such as an integer, a character, a string, a sequence of integers, a set of composite objects, and a set of classes. To allow the function of the process to be specified on an abstract level, one solution is to define the output device as a sequence of the values of a special union type that contains any possible value. We use the keyword **universal** to denote the special union type. Thus, the type declaration

    A = **universal**

defines that type A contains any possible value of any possible type, and the variable declaration

    v: A

allows variable v to take any possible value.

12.3 Is Function

When writing specifications there may be a situation that requires a precise type of a given value. Such a type can be determined by applying a built-in function, known as **is** function. The format of the **is** function is

is_T(x)

The keyword **is** must be used when forming such a function. This function is in fact a predicate that yields **true** when the type of value x is T; otherwise, it yields **false**. Consider the function application:

is_string("Hosei University")

It yields **true** because value "Hosei University" belongs to type **string**. Since this value is also a value of type Mixture, we have the following result as well:

is_Mixture("Hosei University") <=> **true**

The **is** function is applicable to values of any types available in SOFL. Sometimes, a value like 5 can be a value of different types, such as **nat0**, **nat**, **int**, and **real**; the application of the **is** function to all the types on value 5 yields the truth value **true**. For example,

is_nat0(5) <=> **true**
is_nat(5) <=> **true**
is_int(5) <=> **true**
is_real(5) <=> **true**

12.4 A Specification with a Union Type

We take the identifier of SOFL as an example to illustrate the use of union types. An identifier is defined as a string of characters, including English letters, digits, and the underscore mark, but the first character must be an English letter. The type Identifier, whose values have a structure conforming to this restriction, is defined as follows:

Identifier = EnglishLetter | EnglishLetter * IdentifierBody
IdentifierBody = EnglishLetter | Digit | Underscore |
                 IdentifierBody * IdentifierBody
EnglishLetter = {<a>, <b>, <c>, ..., <X>, <Y>, <Z>}
Digit = {<0>, <1>, <2>, <3>, <4>, <5>, <6>, <7>, <8>, <9>}
Underscore = {<_>}

The union type Identifier is composed of type EnglishLetter and the product type EnglishLetter * IdentifierBody that is defined in terms of type English-Letter and IdentifierBody. The type IdentifierBody is then defined recursively as the union type of EnglishLetter, Digit , Underscore, and the product type IdentifierBody * IdentifierBody. Types EnglishLetter, Digit, and Underscore are defined as the enumeration types. Formally, all the 52 English letters (both lower case and upper case letters) of the enumeration type EnglishLetter need to be given in the list, but for brevity, we use ... to denote the omitted letters in the type. As we have explained before, despite a value of an enumeration type being contained in a pair of angle brackets, the brackets do not add any additional meaning to the value. For example, we should understand <c> as letter c rather than the string of the three characters <, c, and >, and <3> as digit 3 rather than the string of the three characters <, 3, and >.

The values of type Identifier can be derived from this type definition. For example, the following are possible values of this type:

mk_Identifier(<a>)
mk_Identifier(<b>, <1>)
mk_Identifier(<b>, **mk_IdentifierBody**(<2>, <3>))
mk_Identifier(<x>, **mk_IdentifierBody**(<_>, <3>))

These values represent the strings of characters, regardless of their syntax. So they can be more intuitively interpreted as the following strings:

a
b1
b23
x_3

12.5 Exercises

1. Define a union type School with the constituent types ElementarySchool, JuniorHighSchool, HighSchool, and University, assuming that all the constituent types are given types.
2. Let s1 and s2 be two variables of the type set of Mixture. Let s1 = {"Hosei University", {3, 5}, 'b'} and s2 = {"SOFL", 'a', 'b', {9}}. Evaluate the following expressions:
   a) **card**(s1) = **card**(s2)
   b) **union**(s1, s2)
   c) **inter**(s1, s2)
   d) **diff**(s1, s2)

3. Let a, b, c: Identifier. Evaluate the following expressions:
   a) **is_**Identifier(a)
   b) **is_**Digit(b)
   c) **is_**EnglishLetter(c)

13

Classes

We have discussed all the built-in types so far, such as basic, set, sequence, composite, product, map, and union types, but these types may not be powerful enough for the construction of large scale specifications due to the limited operators provided in each type. It is often the case that one needs to build one's own types on the basis of the built-in types to provide more flexible and powerful operations over the values contained in the types. Such a user-defined type is known as *class*.

The aim of this chapter is to introduce class, an fundamental concept in object-oriented methods for software development [86], and the other related concepts. These concepts are intended mainly to help construct explicit specifications in SOFL. We start by defining the concepts of class and object, and then proceed to discussions of other important issues in object-oriented design and programming languages [18][22], such as *object identity, access control, inheritance, polymorphism,* and *generosity*.

13.1 Classes and Objects

A class is a user-defined type, which defines a collection of *objects* with the same *features*. The features of objects include *attributes*, describing their data resources, and *operations* offering the means for manipulating their data resources and providing functional services for other objects. An object is an instance of a class with a unique *identity*.

For example, Student is a class that contains a set of specific students, as illustrated in Figure 13.1. Mike, Jean, and John are three students of the class; each of them has attributes id (identification number) and dept (department) he or she belongs to, and can perform the operations Study and Take_exam. These attributes and operations are defined in class Student, which imposes a specification requiring that all the objects in the class have these attributes and operations, but the attributes have no concrete values until an object is instantiated. For example, object Mike has 001 as its id and CS (Computer
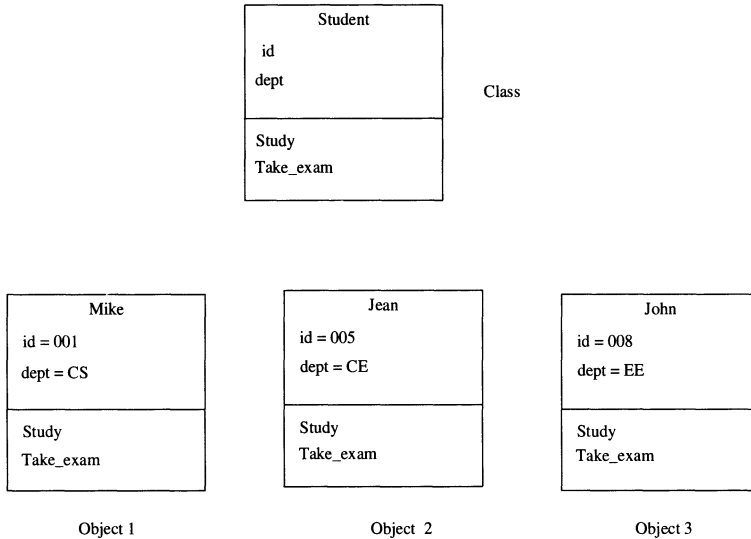
Fig. 13.1. An illustration of class and objects

Science) as its dept; Jean has 005 as its id and CE (Computer Engineering) as its dept; and John has 008 as its id and EE (Electronic Engineering) as its dept.

13.1.1 Class Definition

To use a class as a user-defined type for creating objects, the class must be first defined. The structure of a class takes a format similar to that of a module, introduced in Chapter 4, except the following differences:

- The keyword **class** is used instead of **module** to make a clear distinction between the two similar but different concepts.
- The process specification is replaced with method specification, so the keyword **method** is used instead of **process** for consistency with the convention of terms used in the object-orientation. A method is allowed to have only one or no output, and to have only one port for both input and output.
- The **behav** part in a class is optional: a CDFD can be drawn for improving the readability of the class, but can be omitted if unnecessary.
- A method can be defined either by an implicit specification or explicit specification, but cannot be decomposed into another lower level CDFD. Note that not any method can be defined using implicit specifiction, it may cause confusion in some cases. In SOFL there is a strict rule for defining methods using pre and postconditions: only those methods that do not change objects possibly received from either input or external varaibles of

the methods can be defined using pre and postconditions. In other words, the methods that may change the objects received from input or external variables must be defined as explicit specifications.

Thus, a class in general has the following structure:

class *ClassName / SuperClassName;*
const *ConstantDeclaration;*
type *TypeDeclaration;*
var *VariableDeclaration;*
inv *TypeandStateInvariants;*
behav CDFD_*no;*
method Init;
method_1;
method_2;
    ...
method_n;
function_1;
function_2;
    ...
function_m;
end_class;

A class must have a name, like Student explained in the previous example, and may have a *superclass*, denoted by *SuperClassName*; a slash / is used to separate the class name and the superclass name. The notion of super class will be discussed in Section 13.4 in detail. The variables declared in the **var** section are known as *attribute variables*, which will be used to denote attributes of objects of the class. The method Init is the constructor that has a function similar to the initialization process Init of a module: to initialize the attribute variables when an object is instantiated from the class. Since method Init is supposed to initialize all the attribute variables of the class, the listing of all the attribute variables as its external variables can be omitted for conciseness. A method has the same structure as that of a process, except the constraints given above. In fact, both methods and processes define operations, so they have no substantial differences. However, since the ways of using processes and methods are quite different, which reflects different development paradigms, they are deliberately distinguished by different terms. The principle for adopting the terms is to be as consistent as possible with the term conventions being adopted in both the structured method and object-oriented method communities. Modules and processes are used for constructing a specification in a model-oriented fashion, while classes and methods are employed to construct specification components used by other parts of the entire specification in an object-oriented manner. Both approaches help each other in the construction of large-scale systems.

All the other parts in the outlined structure of a class, such as **const**, **type**, **inv**, **behav**, and **function**, are interpreted in exactly the same way as those of a module. However, we must bear in mind that the declarations of constants, types, invariants, CDFD, and functions are directly associated with the class itself, whereas attributes and methods are associated with specific objects when they are created. That is, the attributes and methods of objects must be accessed through the reference of the objects rather than through the reference of the class from which the objects are instantiated, as we will discuss in detail in the next section.

Now let us take class Student, given previously as an example, to illustrate the definition of class.

class Student;

type

Record = **string** * **real**; /* A record a pair: (course title, score). */

var

id: **nat0**;
dept: Dept;
study_records: **set of** Record;
/* All the records are different becuase of their different id */

inv

id $<=$ 9999;

method Init ()
post id $= 0$ **and** dept $= <CS>$ **and** study_records $= \{ \}$
end_method;

method Study(course: string)
ext wr study_records;
post study_records $=$ **union**(˜study_record, \{mk_Record(course, 0)\})
end_method;

method Take_exam(course: string) score: **real**
ext wr study_records;
post (exists[x: nat0] | 0 <= x <= 100 **and** score = x) **and**
    study_records = **union(diff**(~study_records,
        {**mk_**Record(course, 0)}), {**mk_**Record(course, score)})
end_method;

end_class;

The two attribute variables id and dept of class Student are declared with
types **nat0** and {<CS>, <CE>, <EE>}, an enumeration type, respectively.
The invariant of the class requires that the id of every object be less than
or equal to 9999 for administrative reasons. The constructor Init is a special
method: it initializes the attribute variables of the class when an object is
instantiated. The two methods Study and Take_exam are defined in the class.
Study takes a course as input and registers the course title course, together
with the score 0 (the status before an examination), in the attribute (external)
variable study_records. The method Take_exam takes a course as input and
provides a score as the result of the method. In addition, it also updates the
attribute variable study_record by replacing the pair **mk_**Record(course, 0)
with the pair (course, score) in study_record.

13.1.2 Objects

An object of a class is instantiated from a class through the operator **new**.
To hold an object, we need to have a variable of the class declared,

    obj: C;

where C is a class, presumably defined somewhere else in the specification,
and obj is a variable of class C. An object to be held by the variable obj is
then instantiated in the form

    obj := **new** C;

The attributes of object obj are initialized by the constructor Init() of class C.
For simplicity, we allow a class to provide *only one* constructor named Init.
The definition of the constructor Init() cannot be omitted if the class has at
least one attribute variable and can be omitted if there is no attribute variable
defined in the class.
    Let us consider the class Student as an example. Suppose we declare

    s: Student;

then, we derive an object by applying the **new** operator to the class Student:

    s := **new** Student;

According to the specification of constructor Init() of the class Student given in the previous section, object s satisfies the following properties:

    s.id = 0 **and** s.dept = <CS> **and** s.study_records = { }

where s.id, s.dept and s.study_records refer to the attributes id, dept, and study_records of the object s, respectively.

13.1.3 Identity of Objects

After an object is instantiated, it is assigned to a unique identity distinguishing it from other objects, and this identity is sustained throughout the "execution" of the entire specification. This is similar to the situation that a student is given a unique identification number after he or she enters a university and the number is kept in use until he or she graduates from the university.

For the people who either write or read a specification, there is no need to know what exactly the identities of objects are. It would be sufficient to know that every object created is different, and kept alive until the termination of the execution of the entire system.

13.2 Reference and Access Control

After an object of a class is created, its attributes and methods can be used by other objects in their methods or by processes of a CDFD in their specifications. Suppose obj is a variable of the class C in which attribute variables a_1, a_2, ..., a_n and methods m_1, m_2, ..., m_q are declared and defined. Formally, obj is declared as

    obj: C;

where class C is defined as

    class C;

    var

      a_1: T_1;
      a_2: T_2;
      ...
      a_n: T_n;

```
m_1(...);
m_2(...);
 ...
m_q(...);
end_class;
```

where T_i (i =1..n) are types or classes. Then we refer to the attributes and methods of the object in the form

obj.a_i

and

obj.m_j(...)

where i = 1..n and j = 1..q.

To achieve information hiding, all the attributes of an object must be *private*, that is, the attributes can only be accessed and updated directly by the methods of the object. Any access or updating of the attributes of an object by other objects must be done by means of the methods of the object. When an attribute a_i of an object obj needs to be used for evaluation of an expression or definition of other variables in another object, in principle a method whose function is to get the attribute needs to be invoked. However, to simplify the expression of referring to the attribute and to reduce the effort of defining methods to get attributes of object obj, we use the expression obj.a_i as a shortcut for the method invocation whose result is to yield a_i of object obj. Thus, the reference obj.a_i must not be used on the left hand side of an assignment operator := in an explicit specification outside the class of object obj, but it can be used either on the right hand side of an assignment statement (e.g., obj.a_i may be used in the expression E of the assignment statement x := E) or in an implicit specification (e.g., in the precondition or postcondition of a method) to mean the use (or reference) of the attribute a_i.

The form obj.m_j(...) denotes an invocation of the method m_j of the object obj, with the arguments provided in the parentheses. Such an invocation may change some of the attributes of object obj due to execution of the method, and may return some value as output.

If method m_j returns any value as output, it must be used in an appropriate expression that uses the output of this method for evaluation. However, if it does not return any value, the reference of the method obj.m_j(..) must be used as an independent statement in the explicit specification of a method or process. Since the invocation obj.m_j(...) may cause the change of the

attributes of object obj, such an invocation must *not* be used in implicit specifications, either in the pre or postcondition of a process or method; it can only be used in an explicit specification of a method defined in a class or of a process defined in a module.

For example, a class A is defined as

class A;

var

s: Student;

method Init()
explicit
 s := **new** Student;
end_method;

method Check_Score(course: **string**) exam_score: **nat0**
ext wr s
explicit
 exam_score := s.Take_exam(course);
end_method;
end_class;

In this class, the attribute variable s is declared with class Student, and a method Check_Score is defined by an explicit specification. The variable s is initialized using the **new** operator in the explicit specification of the constructor Init. The execution of the method Check_Score will assign the score, resulting from the invocation of the method Take_exam of object s, to the output variable exam_score of the method Check_Score. Note that in order to ensure that the output variable of a method is defined by its explicit specification, there must exist at least one assignment statement that assigns an expression to the output variable, as it does for exam_score in the above example.

13.3 The Reference of a Current Object

When writing the specification of a method of a class, we may encounter the situation where some parameters of the method share the same name with some attribute variables of the class. If both the parameters and the attribute

variables need to be used in the specification of the method, a confusion in distinguishing the two variables will occur. To solve this problem, we use the keyword **this** to denote the current object, and reference the attribute variable through the current object **this**. Let us look at a very simple example below.

```
class A;
var
 x1: int;
 y1: real;
 ...
 method Add(x1: int, y1: real)
 ext wr this.x1
     wr this.y1
 explicit
  if x1 > 5
  then this.x1 := x1 + y1
  else this.y1 := x1 + y1
end_method;
 ...
end_class
```

Class A has two attribute variables, x1 and y1, which share the same names as the two parameters of method Add. In the specification of Add, **this**.x1 and **this**.y1 refer to the attribute variables x1 and y1, respectively, while x1 and y1 refer to the two parameters of Add, respectively.

13.4 Inheritance

13.4.1 What Is Inheritance

It is quite possible that many classes share the same characteristics (attributes and methods), but some classes have fewer attributes and/or methods than other classes. The classes with fewer attributes and/or methods may be used as the basis to build the other classes with extended attributes and/or methods. For example, we can build another class known as Student_with_Scholarship that has all the attributes defined in class Student discussed previously and another additional attribute scholarship. Thus, a student with scholarship, an object of class Student_with_Scholarship, should also be a student (object) of class Student. In this case, we say class Student_with_Scholarship inherits from class Student. Thus, inheritance serves as a mechanism for building new classes based on existing classes, and therefore allows for the reuse of attributes and behaviors (methods) of some classes by some other classes. For example, we can define class Student_with_Scholarship as follows:

```
class Student_with_Scholarship / Student;


var


scholarship: int; /* amount of the money provided by the scholarship */


method Init()
post scholarship = 0
end_method;
end_class;
```

This definition explicitly indicates that class Student_with_Scholarship inherits from class Student by providing the class name Student after the slash symbol /. Apart from the additional attribute scholarship, this class also treats the attributes, id and dept defined in class Student as its own attributes and the methods Study and Take_exam as its own methods.

13.4.2 Superclasses and Subclasses

In general, if class B inherits from class A, we define B in the form:

```
class B / A;
...
end_class;
```

Thus, class B inherits all the attributes and methods defined in class A. In fact, by using this mechanism we are able to construct a class inheritance hierarchy: all classes lying below a class inherit from the class. As an example, we define another two classes C and D, both inheriting from class B:

```
class C / B;
...


end_class;

class D / B;
...


end_class;
```
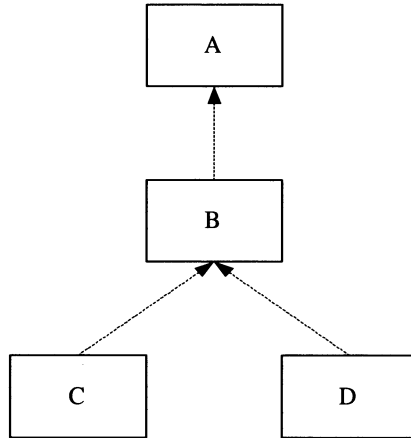
Fig. 13.2. An inheritance hierarchy

Thus, C inherits all the attributes and methods defined in both B and A. From the object point of view, an object of class C is also an object of classes B and A, but not vice versa.

Figure 13.2 illustrates the inheritance hierarchy composed of these four classes. Since an inheritance hierarchy may include several level classes, we may want to emphasize the relation between two classes. If a class directly inherits from another class, we call this class a *direct subclass* and the class a *direct superclass*. For example, class B is a direct subclass of A, while A is a direct superclass of B. Any class lying below a class in the inheritance hierarchy is known as a subclass of this class, while this class is known as a superclass of the class lying below it. For instance, class C is a subclass of A and A is a superclass of C, but they are not a direct subclass and superclass of each other, respectively.

Note that SOFL does not support *multiple inheritance*, that is, we do not allow a class to inherit from more than one superclass. There are two reasons for this. One reason is that, by enforcing the single inheritance, we can avoid the difficulties in dealing with multiple superclasses with homonymous properties (e.g., the same name for different attribute variables or methods in different superclasses) and in verifying specifications (e.g., reviewing and testing specifications). Another reason is that a multiple inheritance can be converted to a single inheritance through good design. Due to this restriction, it will not be difficult at all to transform a SOFL specification into a program of an object-oriented programming language allowing only single inheritance, like Java.

13.4.3 Constructor

An object of a class in an inheritance hierarchy is instantiated by executing its constructor. If this class inherits from its direct superclass (if any), the unique constructor Init of its direct superclass is assumed to be automatically executed at the beginning of the execution of its own constructor. There is no need to write anything explicitly in the definition of the subclass constructor to indicate that the constructor of its superclass is invoked. This assumption is part of the semantics of SOFL; it should not be forgotten when interpreting a specification involving class hierarchies.

For example, suppose an object of class C is instantiated and held in variable obj1, which is written as

```
obj1: C;
obj1 := new C;
```

which invokes the constructor Init of class C. At the beginning of the execution of this constructor, the constructor Init of class B is assumed to be invoked in order to initialize all the attributes defined in class B; these initialized attributes are part of the attributes of the object of class C being created.

13.4.4 Method Overloading

Method overloading is a way to define different methods with the same name. SOFL allows several methods of the same name to be defined as long as these methods have different sets of parameters (based on the number of parameters, the types of the parameters, and the order of the parameters). When an overloaded method is invoked, the number, types, and order of the arguments of the method in the invocation will be compared to those of the corresponding parameters of the method in its definition, and the completely matched one will be executed. Method overloading is commonly used to create several methods with the same name that perform similar tasks, but on different data types. For example, in the class Exam we define the overloaded method square to calculate the square of an integer and the square of a real number, respectively.

```
class Exam;
  ...
method square(x: int) res: int
post res = x ** 2
end_method;

method square(x: real) res: real
post res = x ** 2
end_method;
end_class;
```

However, for simplicity, the principle of method overloading is not applicable to the constructor Init of a class, that is, we do not allow two or more overloaded Init to be defined in the same class. If one wants to initialize some attribute variables with specific values, one can define a method to update the initialized attribute variables properly with the specific values.

13.4.5 Method Overriding

Method overriding is a mechanism that allows a subclass to redefine a method of its superclass while sustaining its interface. Let us take the class hierarchy given in Figure 13.2 as an example. Let a method m1 be defined in class A. Then it is inherited by class B, as B is a subclass of A. However, for some reason, the definition of method m1 may need to be modified. In this case, the following rules must be followed:

1. The interface of method m1 must be sustained: the name, input parameters, and output parameters of method m1 cannot be changed.
2. Every other part of method m1 is subject to change.

Consider the extensions to classes A and B:

class A;
...

method m1(x: int) y: int
post y ** 2 $=$ x
end_method;

method m2()
...
end_method;
end_class;

class B / A;
...

method m1(x: int) y: int
post y ** 2 $=$ x and y $>$ 0
end_method;
end_class;

Let b be an object of class B. Then, the invocation of method m1 in the form

b.m1(5)

refers to the method m1 defined in class B instead of the corresponding method
in class A. It is a general principle that a method invocation of any object
refers to the corresponding method definition given in its own class. If the
invoked method is not defined in its own class, however, the invocation will
search for the method definition given in the direct superclass of the current
class, and then another higher level superclass, and so on, until the method
definition is found in the class hierarchy in a bottom-up manner. For example,
the method invocation

b.m2()

refers to the definition of method m2 given in class A, although b is an object
of class B, because m2 is not defined in class B, and B is a direct subclass of
class A.

13.4.6 Garbage Collection

Garbage collection is an operation that collects the memory spaces occupied
by lost objects (the objects no longer being used) during the execution of
a program written in an object-oriented programming language (e.g., C++,
Java). Since SOFL is a specification language that does not concern itself
with memory issues, there is no need to address the garbage collection is-
sue in specifications. However, when a specification is implemented using an
object-oriented programming language, the garbage collection issue must be
addressed in order to achieve efficiency in the use of computer memory.

13.5 Polymorphism

Polymorphism is a mechanism by which a single method may be defined in
more than one class and may take on different implementations in each of those
classes. It is usually implemented dynamically on the basis of inheritance of
classes. For example, if a variable of a class is used as an input variable to
a method, then it may be bound to different objects coming from the same
class as that of the variable or from its subclasses. For this reason, the method
may behave differently, because different objects may use different methods
(in terms of their functionality) that may share the same interface.

Let us take classes A and B, declared previously, as an example, where B
is a subclass of A. Suppose method d is defined in class D as follows:

```
class D;
...

method d(x: A)
explicit
x.m1(5);
...
end_method;
...
end_class;
```

Then, when method d is invoked, its input variable x can be bound to objects of either class A or class B. This is because objects of B can be treated as objects of A due to the inheritance relationship between B and A. However, if the signature of method d is changed to

```
method d(x: B)
explicit
x.m1(5);
...
end_method;
```

then, since objects of class A are not regarded as objects of class B, invoking method d with an object of class A will be disallowed. This point is similar to the situation where a variable of type **real** can be bound to either real numbers, integers, or naturals, as both integers and naturals are real numbers, but not vice versa.

Now, let us look at why method d could behave differently when x is bound to an object of class A or B. If x is bound to an object, say a, of class A, x.m1(5) means the invocation of method m1 defined in class A. In other words, it is equivalent to the invocation a.m1(5). However, if x is bound to an object, say b, of class B, x.m1(5) means the invocation of method m1 defined in class B, that is, it is interpreted as b.m1(5). As the same method m1 may be defined differently in class A and B, the invocation of x.m1(5) in method d may behave differently, which may result in a different behavior of method d itself.

An operator known as *downcast* allows us to convert an object of a class to that of its subclass. Such an operator is given in the form

```
(className)
```

where className is the name of the class to be converted to. When applying this operator to an object, say x given in the previous example, of class B, the object x is converted into an object of class B. For example,

```
(B) x;
```

Thus, the invocation of methods defined in class B can be carried out by means of object x. To avoid any potential confusion in interpreting expressions involving the downcast operator, we specify that the downcast operator has the highest priority of application. For example, the method invocation

(B) x.m1(5);

means that the downcast operator (B) is first applied to object x, and then the method m1 of the resulting object is invoked. A clearer expression may be obtained by using parentheses,

((B) x).m1(5);

The built-in boolean function *is-function*, introduced in Chapter 12 can also be applied to determine whether an object belongs to a specified class. For example,

is_A(a) <=> **true**
is_B(a) <=> **false**
is_B((B) a) <=> **true**
is_A(b) <=> **true**
is_B(b) <=> **true**

13.6 Generic Classes

A class is a generic class if it allows type parameters that will be bound to concrete types (or type identifiers). The parameters are used as types to declare variables within the class definition, and must be bound to specific types when variables are declared using the class.

Let A be a generic class with type parameter T. Then, A is declared in the form

class A[T];

...

end_class;

For example, the type parameter T is used to declare a state variable s1 and the input parameter of method m:

```
class A[T];

var

s1: seq of T;

method m(s: set of T)
...
end_method;
...
end_class;
```

Semantically, a generic class represents a mapping from a set of types to a set of classes. Consider class A[T] as an example, it is defined as

A[_]: Types -> Classes

where instances of T are members of Types, that is, the collection of all possible types, and the class of A[T], derived by binding a specific member of Types to T, is a specific class of Classes denoting the collection of all possible classes.

When a variable x is declared with class A, it is necessary to specify a concrete type, say **real**, for T. Thus, a declaration of x can be

x: A[**real**];

The state variable s1 in class A then becomes a variable of type **seq of real**, and the parameter s will be bound to values of type **set of real**.

Note that type parameter T of class A can also be bound to another class, but *not another generic class* because a generic class cannot be treated as a concrete type. Of course, if a language allows high-order functions, such a high-order class may be allowed. However, it is the principle of SOFL that the first order logic be a good balance between the expressive power and the simplicity for practice.

The principle of generic class described above can be extended to multiple type parameters of classes. That is, we allow a class, say B, to be declared in the form

```
class B[T_1, T_2, ..., T_n];
...
end_class
```

Thus, a variable y of class B may be declared as follows:

y: B[**int, real, set of nat0**]

13.7 An Example of Class Hierarchy

In this section, we describe in detail how to build a class inheritance hierarchy by discussing an example of building classes Point, Circle, and Cylinder. Since any object of a point, circle, and cylinder has a center point, class Point is a sensible generalization of Circle and Cylinder. Therefore, we first define Point as a class, and then define Circle as a subclass of Point, and Cylinder as a subclass of Circle.

Since a point consists of coordinates x and y, they should be declared as the attributes of class Point. To create points at required coordinates x and y, we define a method known as setPoint. Thus:

```
class Point;


var


 x, y: int;


method Init()
post x = 0 and y = 0
end_method;


method Set_Point(a: int, b: int)
ext wr x, y
post x = a and y = b
end_method;


end_class;
```

Now we define Circle as a subclass of Point:

```
class Circle / Point;

const


PI = 3.14;

var
```

```
radius: real;   /* radius of Circle */

method Init()
post radius = 0.0
end_method;


method Set_Radius(r: real)
ext wr radius: real
pre r >= 0.0
post radius = r
end_method;


method Create_Circle(x1, y1: int, r: real)
ext wr x, y: int
    wr radius: real
pre  r >= 0.0
explicit
  Set_Point(x1, y1);
  radius = r;
comment
```
This method is defined with a mixed specification of implicit and explicit styles. The precondition requires that parameter r be greater than 0.0, while the explicit specification defines how to create a circle. Note that attributes x and y inherited from class Point are accessed through method Set_Point of class Point, while attribute radius is directly assigned a value because it is defined in the current class Circle.
```
  end_method;


method Compute_Area() area: real
ext rd radius
post area = PI * radius ** 2
end_method;


  end_class;
```
Since class Circle inherits from class Point, there is no need to carry out explicitly an initialization of the inherited attributes x and y in the specification of the constructor Init of class Circle, because it is assumed to be implicitly done through the Init method of the superclass Point by SOFL semantics.

The method Set_Radius and Compute_Area are simple; the former sets the attribute radius with a new value r, while the latter computes the area of the current circle object. In the specification of Compute_Area, the constant

PI, which is defined in the **const** part of the class, is used in the definition of the area of the circle.

The most interesting method in this class is Create_Circle; it shows a need for using a mixed specification of implicit and explicit styles. When defining conditions for variables of the built-in types (e.g., **int**, **real**, **string**, **seq of real**), pre and postconditions can be used, but when defining conditions or functions involving objects (e.g., an object of Point), explicit specification must be adopted (in this case, precondition can still be defined if it involves only variables of the built-in types).

When writing an explicit specification for a method, since the primary concern is how the behavior of the method is provided by means of the invocation of methods of the related objects, we must enforce the principle that attributes of a class can only be dealt with by means of its methods. For example, the method Set_Point(x1, y1) is invoked to change the attributes x and y of the current object of the class Circle, instead of directly using the assignments x := x1 and y := y1, because both attributes x and y as well as the method Set_Point are declared in the class Point. In contrast with this, attribute radius is defined using the assignment statement: radius := r, since it is an attribute of the current object of the class Circle.

It is possible to provide both an implicit and an explicit specification for the same process, although there is no need to do so in general. In fact, implicit specification is designed for *abstract design*, while explicit specification is for *detailed design*. However, this is not a definitive rule for the use of implicit and explicit specifications; they should be utilized with flexibility, depending on the concrete application. Sometimes it may be easier to describe the behavior of a method in an explicit manner, while at other times it may be simpler to describe the behavior using an implicit specification. The fundamental point is that *an explicit specification is needed if invocations of methods of related objects are inevitably used; otherwise, an implicit specification may be written.*

As a subclass of Circle, the class Cylinder is defined as follows:

```
class Cylinder / Circle;
var
 height: real;  /* height of Cylinder */
method Init()
post height = 0
end_method;
method Set_Height(h: real)
ext wr height
pre h >= 0.0
post height = h
end_method;
method Create_Cylinder(x1, y1: int, r, h: real)
ext wr x, y: int
    wr radius: real
```

  wr height
 pre r $>=$ 0.0 **and** h $>=$ 0.0
 explicit
  begin
  Set_Point(x1, y1);
  Set_Radius(r);
  Set_Height(h)
 end
 end_method;
 method Compute_Surface_Area() area: **real** /*surface area of Cylinder */

 ext rd radius: **real**
 explicit
 begin
  area := 2 * Compute_Area() + 2 * Circle.PI * super.radius * height
  /*the constant PI defined in the class Circle is used. */
 end
 end_method;
 method Compute_Volume() vol: **real** /*surface area of Cylinder */
 ext rd height: **real**
 explicit
 vol := Computer_Area() * height;
 /* a method of the superclass is invoked. */
 end_method;
 end_class;

Since the evaluation of the surface area of cylinder contains the invocation of the method Compute_Area of the superclass Circle, an explicit specification is adopted for defining the method Compute_Volume. Compute_Area is defined as a public method in class Circle and inherited by class Cylinder; it is therefore used without referring to the superclass. In the method Compute_Surface_Area, we need to refer to the private attribute variable radius of the superclass in order to compute the surface area of the cylinder object. We use super to represent the object of the direct superclass from which the current class inherits, and therefore we use super.radius to refer to the attribute radius of the related object of the direct superclass.

13.8 Example of Using Objects in Modules

The example presented in this section aims to explain how classes, objects, and their methods are used in building modules, including CDFDs and process specifications. After building class Cylinder in the previous section, we now want to build a module whose CDFD describes the behavior of creating an object of Cylinder and carrying out some interesting evaluations, as illustrated in Figure 13.3. The center point of the bottom circle of the cylinder is created
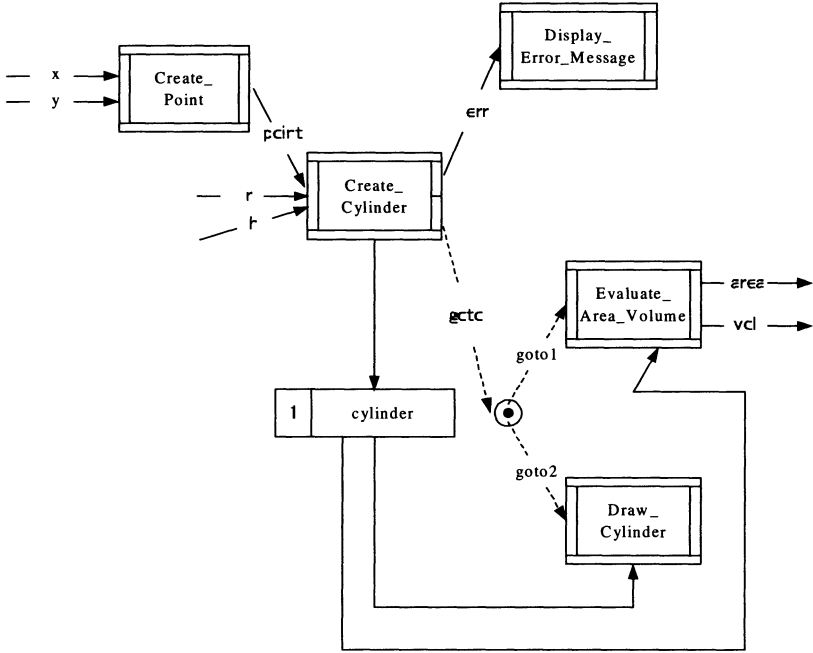
Fig. 13.3. A module using objects of classes

by process Create_Point based on the input coordinates x and y. Then, store cylinder, an object of class Cylinder, is created by process Create_Cylinder, based on the inputs point, r (radius), and h (height). If radius r or height h is not positive, error message err is produced; otherwise, control data flow goto is made available. The copies of this data flow, goto1 and goto2, are then used as the inputs of processes Evaluate_Area_Volume and Draw_Cylinder, respectively. Process Evaluate_Area_Volume calculates the surface area and volume of the cylinder, while Draw_Cylinder draws the cylinder on an output device (e.g., screen).

The specification of the module whose behavior is described by the CDFD in Figure 13.3 is given below. For constructing this module, the classes Cylinder and Point are assumed to have been defined previously. Thus, they can be directly used in the module.

```
module Cylinder_Test;
var
cylinder: Cylinder;
   /*class Cylinder is assumed to have already been defined.*/
process Init()
explicit
 cylinder := new Cylinder
```

```
comment
This process initializes the only store variable, cylinder.
end_process;
process Create_Point(x, y: int) point: Point
explicit
begin
  point = new Point;
  point = point.Set_Point(x, y)
end
comment
A point is created by the invocation of method Set_Point of
object point.
end_process;
process Create_Cylinder(point: Point, r, h: real) err: string | goto: sign
ext wr cylinder
explicit
  if r >= 0.0 and h >= 0.0
 then
  begin
     cylinder := cylinder.Create_Cylinder(point.x, point.y, r, h);
     goto := !;   /* making goto available */
   end
  else
     err := "The radius or height of the cylinder is negative"
comment
This process creates an object cylinder of class Cylinder by invoking method
Create_Cylinder of cylinder based on the inputs point, r, and h, if the radius
r and the height h are both positive; otherwise, the error message err is
generated.
end_process;
process Display_Error_Message(err: string)
post Display(err)
comment
The error message err is displayed on an output device by means of the
function application Display(err). The Display function is defined later in this
module.
end_process;
process Evaluate_Area_Volume(goto1: sign) area: real, vol: real
ext rd cylinder
explicit
begin
 area := cylinder.Compute_Surface_Area();
  vol  := cylinder.Compute_Volume()
end
comment
```

The surface area and vol are generated by assigning the results of invoking the methods Compute_Area and Compute_Volume of the object cylinder, respectively.

end_process;
process Draw_Display(goto2: **sign**)
ext rd cylinder
post Draw(cylinder)
comment
The drawing of cylinder is done by the function application Draw(cylinder). The function Draw is defined below.
end_process;
/* Display is intended to display the given error message on an output device */
function Display(meg: **string**): **bool**
== **undefined**
end_function;
/* Draw is intended to draw the given cylinder on an output device */
function Draw(cylinder: Cylinder): **bool**
== **undefined**
end_function
end_module.

Since this module is built by applying the knowledge of SOFL we have introduced before, there is no need to explain this specification further, except the point of defining the functions Display and Draw with the keyword **undefined**. The functions defined with the keyword **undefined** indicate that we do not care in the phase of formal specification how the error message is displayed and how the cylinder is drawn on an output device, because it is an implementation problem. However, if this point is a primary concern in either requirements or design, it should be defined precisely in the specification.

13.9 Exercises

1. Answer the following questions:
   a) What is a class?
   b) What is an object?
   c) What is inheritance?
   d) What are superclass and subclass?
   e) What is polymorphism?
   f) What is a generic class?
2. Define the class Polygon as the superclass of the classes Triangle and Rectangle. Define an attribute variable area and a method Compute_Area in each of the classes, but with different specifications, depending on the specific shapes.

3. Specify a module whose CDFD creates a required shape that can be one of the objects of the two classes **Triangle** and **Rectangle**, and compute its area.

14

The Software Development Process

The development of a complex software system needs a well-planned process to ensure its quality. There are already many existing process models, such as the *waterfall model* [32], the *spiral model* [106], the *formal methods-based transformation model*, and the *prototyping model* [84], and each model indicates the necessary activities, resources, relations between activities and resources, and necessary technologies for carrying out the activities. Although each of these models has a different emphasis on the way systems are developed, they all share the commonality of using the waterfall model as the underlying concept.

14.1 Software Process Using SOFL

Developing software systems with SOFL takes a combined approach of the waterfall and transformation models. The overall process shares the similarity with the waterfall model of emphasizing the activities of requirements analysis, design, and coding, but differs from the conventional transformation model in the way that transformations from high level documentation into low level documentation may not necessarily be strict refinements; it can be either an *evolution* or refinement, depending on the phase of the development. Feedbacks from later phases to early phases are usually inevitable, and such feedbacks may lead to changes of the documentation produced in early phases.

Figure 14.1 illustrates the SOFL process model. One of the important features of this model is to support the *three-step approach* for building formal specifications: *informal*, *semi-formal*, and *formal specification*. Each specification contains different level information of the requirements and serves a different role. In this chapter we concentrate on the explanation of the three-step approach to specification construction, with a simple example, and discuss briefly the related activities in the process model. The specific techniques for specification, verification and validation, and transformation from specifications to programs are described in detail from Chapter 15 to Chapter 19.
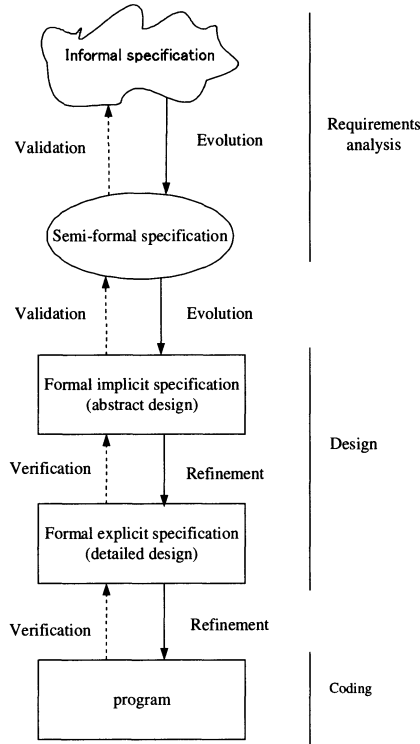
Fig. 14.1. The software development process using SOFL

14.2 Requirements Analysis

Requirements analysis is an activity to discover, understand, and document the user's requirements [51]. It is usually the starting point of building a system, after the feasibility study of the system is completed with a positive conclusion. Since intensive communications between the developer and the end user are frequently involved, the documentation usually has to be written in a natural language, possibly coupled with other comprehensible notations (e.g., diagrams). If the system to be developed is rather complex and has a corresponding system in the real world, such as a banking system, the real world system may first need to be modeled in order to derive accurate and complete requirements. Otherwise, the requirements can be documented informally based on communications between the developer and the user. Such an informal documentation is known as *informal specification.*

Trying to use a formal specification language from the very beginning would cause tremendous difficulties, not only because of the difficulty in communication, but also due to the lack of sufficient knowledge about the requirements. However, this does not mean that the preciseness of the specification is

not important. The point is that we need to strike a good balance between pre-ciseness and readability of the specification. Such a balanced documentation is known as *semi-formal specification*. To achieve this kind of requirements specification, it is natural to take two steps: from informal to semi-formal specifications.

14.2.1 The Informal Specification

Informal specifications are often criticized as a cause of faults in software systems due to the ambiguity in expressions, but in fact the way of writing informal specifications should take the blame even more. A well organized specification can make a significant difference in reducing misunderstanding and complexity of requirements. Although it is difficult to define the concept of *well organized specification*, such a specification must clearly and concisely describe the following items:

- the functions to be implemented
- the resources to be used
- the necessary constraints on both functions and resources

At this stage, there is no need to pay much attention to the relations between functions, resources, and constraints; these will be the task for the stage of semi-formal specification. Let us take a simplified *hotel reservation system* as an example. An informal specification of this potential system is given as follows:

1. The required functions:

- Reserve room
- Cancel reservation
- Change reservation
- Check in
- Check out

2. The resources of the hotel to be managed:

- Single rooms: 100
- Twin rooms: 50
- Double rooms: 50

3. The constraints reflecting the policy of the hotel:

- One customer can reserve only one room each time.
- Only customers with reservations can check into the hotel.

```
1 . T h e  f u r c t i c r s  c f  t h e  s y s t e m :
    ( 1 )  F 1
    ( 2 )  F 2
    ( 3 )  F 3
    ( 4 )  F 4

1 .1  F 1
    ( 1 )  F 1 1
    ( 2 )  F 1 2
    ( 3 )  F 1 3

1 .2  F 2
    ( 1 )  F 2 1
    ( 2 )  F 2 2

● ● ● ● ● ●

2 .  R e c c u r s e s :

3 .  C c r s t r a i r t s :
```
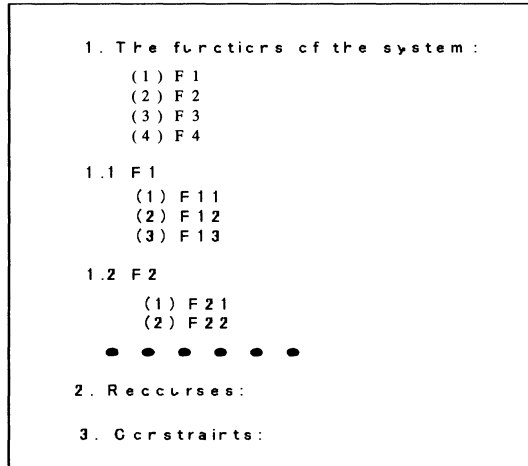
Fig. 14.2. The outline of an informal specification

This informal specification consists of three parts: the required functions, the resources of the hotel to be managed, and the constraints reflecting the policy of the hotel. The functions includes the room reservation, cancellation of reservations, change of reservations, checking in, and checking out services. The room resources which the hotel manages include 100 single rooms, 50 twin rooms, and 50 double rooms. The hotel has a policy requiring that a customer can reserve only one room each time and no customer can check into the hotel without reservation (e.g., for security reasons).

This example only shows an abstract idea of how to organize an informal specification. For a more complex system, some of the functions may indicate a complex task. In that case, those functions may need to be described in detail, indicating their sub-functions. For this purpose, each function can be taken as an abstraction of a module in the specification. There is no need to strictly follow the syntax of the module in this stage, but conceptually each function being treated as a module will help in the creation of the semi-formal specification in the next stage. For example, if function Reserve room needs more detailed description, it can be written as follows:

1.1 Reserve room includes the functions:

● Check the vacancy.
● Register the customer on the reservation list.
● Issue the reservation number.

In a similar way, the other functions in this example can also be decomposed into detailed sub-functions, if necessary. Thus, the entire informal specification may take the form illustrated in Figure 14.2.

An important point about an informal specification is that the specification needs to cover as many functions, resources, and constraints as necessary. In primary and critical functions, especially, resources and constraints must not be missed. In other words, achieving the completeness of requirements in the specification, in terms of the aspects to cover, is essential. Although there is no definitive formula to follow in discovering complete requirements, the field of *Requirements Engineering* has come up with many techniques to help analysts. Since this topic is beyond the scope of this book, we do not develop further along this line.

14.2.2 The Semi-formal Specification

The semi-formal specification derives from the informal specification. Its goal is to clarify and define all the functions, resources, and constraints, and to determine the relations among those three parts contained in the informal specification. For example, which resources should be allocated to which functions, and which constraints should be applied to which functions and/or resources, and so on. To this end, the formal notation can play a positive role. Since the semi-formal specification still serves as a vehicle for communication between the developer and the end user, it should not be fully formal, because the user should not be expected to understand a specific formalism used for writing the specification. The most distinct feature of a semi-formal specification is that *the format of the specification obeys the syntax of the specification language, but the pre and postconditions of all processes in modules are defined or described in a natural language in an organized manner*. This idea is similar to *pseudocode* for program design.

Specifically, the tasks and features of a semi-formal specification should include:

- Associating data resources, constraints, and functions in modules. That is, using modules to encapsulate the data resources, related data constraints, and the related operations that conform to the functional constraints.
- The specification is defined as a set of related modules. Each module defines either a function given in the informal specification or a derived function resulting from further decomposition of an existing function.
- All the data used in the specification are defined with appropriate data types precisely in modules, but their constraints, which are usually given as invariants, are defined in an informal manner. In these definitions, types are allowed to be defined as given types, if necessary.
- The CDFD for each module can be constructed when it is necessary to reflect user requirements for the inter relation of processes in the module. There is no strict restriction of whether CDFDs should be drawn for modules in semi-formal specifications; it all depends on whether it benefits the definition of the user requirements.
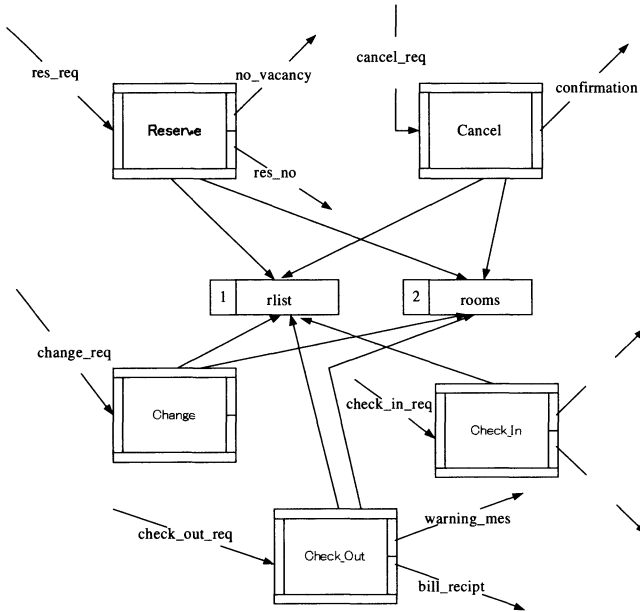
Fig. 14.3. CDFD__1

- All the requirements for operations are defined in terms of processes and functions in the associated module. Such definitions include the declaration of input data flows, output data flows, and external variables representing related stores. They also include the pre and postconditions, but the contents of the pre and postconditions are usually written in an informal manner.

These features emphasize the importance of both grouping data resources, constraints, and operations, and clarifying data structures of all the related data items by defining formally corresponding data types. This is because strategies for performing functions often depend on the data structures of the data items to be processed. Also, by defining the data types the developer can get help in understanding the business of the real world system and the potential functional targets.

For example, a semi-formal specification for the hotel reservation system is constructed. The outline of the specification that helps to explain the features of the semi-formality is given as follows:

    module Reservation_system;
    type

    FullName = **given**;
    Customer = **composed of**
            name: FullName

                 address: **string**
                 tel: **nat0**
                 pass_no: **string**
                 reservation_no: **nat0**
               end;
Room = **composed of**
           room_no: **nat0**
           room_type: {<Single>, <Twin>, <Double>}
           status: {<Reserved>, <Check In>, <Check Out>}
           end;
  ReservationList = **map** Customer **to** Room;
  Rooms = **set of** Room;
  ReservationRequest = **given**;
  CancellationRequest = **given**;
  ChangeRequest = **given**;

 var

  rlist: RservationList;
  rooms: Rooms;

 inv

  (1) every room in rooms is available for reservation;
  (2) the status of every customer on the reservation
     list rlist must be either <Reserved> or <Check In>;

behav CDFD_1;

process Reserve (res-req: ReservationRequest)
                  no-vacancy: **string** | res_no: **nat0**
ext rlist, rooms
pre the customer is not on the reservation list rlist.
post if there is vacancy
     then reserve a room for the customer **and**
        issue a reservation number.
     else produce a no_vacancy message.
end_process;

process Cancel(cancel_req: CancellationRequest)
               confirmation: **string**
ext rlist
pre   the customer is on the reservation list rlist
post (1) delete the customer's reservation from rlist.

      (2) release the reserved room resource by returning
          it to rooms so that it can be used for another
          reservation.
end_process;

process Change(chage-req: ChangeRequest)
                 no-vacancy-mes: **string** |
                 confirmation-mes: **string**
ext rlist, rooms
pre   the customer is on the reservation list rlist
post (1) if the requested room is available, cancel the
         original reservation and make a new reservation.
       (2) if the requested room is not available, generate
         a no vacancy message.
end_process;

process Check_In(customer: Customer)
               no_reservation_mes: **string** |
               room_no: **nat0**
ext rlist
post (1) if the customer has a reservation recorded on
         the reservation list rlist, check in the customer.
       (2) if the customer has no reservation, a message
         of refusing check in is issued.
end_process;

process Check_Out(room_no: **nat0**)
                warning_mes: **string** |
                bill_receipt: **string**
ext rlist, rooms
post (1) if room_no is associated with a check in customer
        on the reservation list rlist, delete the customer's
        information and release the room.
       (1.1) calculate the cost and print out the bill.
       (1.2) receive payment and print out receipt.
      (2) if a room to be checked out is associated with no
        customer on the reservation list rlist, generate a
        warning message.
end_process;
end_module;

In the specification, the module Reservation_System is defined. In this module, seven types are defined: FullName, Customer, Room, ReservationList, Rooms, ReservationRequest, CancelRequest, and ChangeRequest. ReservationList defines

a map type from the composite type Customer to type Room; Rooms is de-
fined as a set type; FullName, ReservationRequest, CancelRequest, and Chang-
eRequest are all given types. These types are then used to declare the store
variables rlist and rooms, and the input or output variables for processes Re-
serve, Cancel, Change, Check_In, and Check_Out. In the invariant part, two
invariants are given to impose constraints on the type Rooms and the store
variable rlist, respectively. In the specification of process Reserve, both pre and
postconditions are given basically in English, but combined with an **if-then-
else** expression for readability. For the other processes, the specifications are
written in English, but possibly in an enumerated manner. Of course, this does
not oppose the use of any useful formal expressions, such as **if-then-else** and
let expressions, together with informal descriptions.

In the development of critical systems, such as safety-critical, security-
critical, and commerce-critical systems, providing formal specifications for
the critical parts in the phase of requirements analysis may be necessary
and cost-effective. But, in general, semi-formal specifications are sufficient for
documenting user requirements.

14.3 Abstract Design

Abstract design transforms the semi-formal requirements specification into a
formal specification that represents *the architecture of the entire system* and
functional definitions of its components. There is a substantial difference be-
tween the requirements specification and the design specification: the former
focuses on the description of the user's requirements, while the latter focuses
on the architecture of the system to provide a solution to the problem. There-
fore, such a transformation is not only a formalization, but also involves the
creation of a system structure that fulfils the requirements.

In general, the system architecture is different from the structure of the
requirements specification, but this does not mean that the semi-formal re-
quirements specification cannot be reused in the design. In fact, it is often the
case that during the construction of the system architecture, some of the mod-
ules, types, processes, and functions in the semi-formal specification may be
employed without any change to their interfaces and functionality, but some
others may need to be modified, extended, or combined with others. In ad-
dition to formalizing the existing semi-formal specification, new specification
components are usually created and integrated into the system to support the
functionality of the entire system.

Formalization of abstract design can benefit the system development in
several ways. Firstly, the designer is given a chance to study the requirements
rigorously and to clarify the ambiguity in the semi-formal specification. This
would force the designer to communicate with the analyst who wrote the
semi-formal specification. In the era of globalization in business and commu-
nication, distributed software projects carried out through the Internet are

increasing. Requirements analysis may be conducted in one place, while the design may be carried out in another remote location. In this situation, formalization of abstract design based on the semi-formal requirements specification definitely helps the designer to study, clarify, and understand the requirements thoroughly. Secondly, the activity of formalization can also stimulate discussions among the developing team members, and therefore it would help to improve their understanding of their tasks. Finally, the design specification serves as a firm foundation for the detailed design, coding, and testing; therefore, it can facilitate the transformation from the abstract design to the detailed design and, further, to the program.

Apart from the criteria imposed to the semi-formal specification, a formal specification is required to satisfy the following additional criteria:

- All the modules are integrated into a hierarchy of CDFDs.
- All the given types are defined precisely. In other words, no given types are allowed in the formal specification, because their values are not defined precisely.
- The pre and postconditions of every process and function in modules are written in the SOFL language, not in any informal language.

Note that these criteria do not forbid the use of explicit specifications for processes, but implicit specifications are encouraged because the focus of this phase is on the architecture of the entire system and the functionality of its components; the issue of how the components are expressed in an algorithmic manner should be left to the detailed design.

In principle, in the stage of abstract design, we do not encourage defining classes and then using their objects in process and function specifications. The reason is that, in the use of objects, invoking their methods may cause undesirable changes of the same object, and therefore cause confusion in pre and postcondition semantics. If methods of an object must be invoked in a process specification in order to define its behavior, an explicit specification must be adopted to define the process. In order to avoid undesired side effects, no object is allowed to be used as a parameter of a function in its specification.

Let us consider the simplified hotel reservation system as an example to see how to transform a semi-formal specification into a formal specification. On the basis of our understanding of the semi-formal specification, a *top-down approach* is taken to design the formal specification. The top level CDFD in the specification aims at dealing with various requests, such as reservation, cancellation, and change of reservation, but takes one at each time. Then, the request is passed to a specific program component to process. Figure 14.4 illustrates the top level CDFD of this system, and the associated module is given as follows:

module SYSTEM_Hotel_Reservation;

type

FullName = **string** * **string** * **string**;
/*first name, middle name, and family name */
Customer = **composed of**
        name: FullName
        address: **string**
        tel: **nat0**
        pass_no: **string**
        reservation_no: **nat0**
      end;
Room = **composed of**
      room_no: **nat0**
      room_type: {<Single>, <Twin>, <Double>}
      status: {<Reserved>, <Check In>, <Check Out>}
     end;
ReservationList = **map** Customer **to** Room;
Rooms = **set of** Room;
RoomNo = **nat0**;

var

ext #rlist: ReservationList;
ext #rooms: Rooms;

inv

forall[x: RoomNo] | 1 <= x <= 200;
forall[x: rooms] | x.status = <Check Out>;
forall[x: **dom**(rlist)] | rlist(x).status = <Reserved> **or**
              rlist(x).status = <Check In>;

behav CDFD_No1;

process Hotel_Reservation(res_req: ReservationRequest |
                  cancel_req: CancellationRequest |
                  change_req: ChangeRequest |
                  check_in_req: Customer |
                  check_out_req: RoomNo)
                  no_vacancy: **string** | res_no: **nat** |
                  confirmation: **string** |
                  no_vacancy_mes: **string** |
                  confirmation_mes: **string** |
                  no_reservation_mes: **string** |
                  room_no: RoomNo |
                  warning_mes: **string** |
                  bill_receipt: **string**
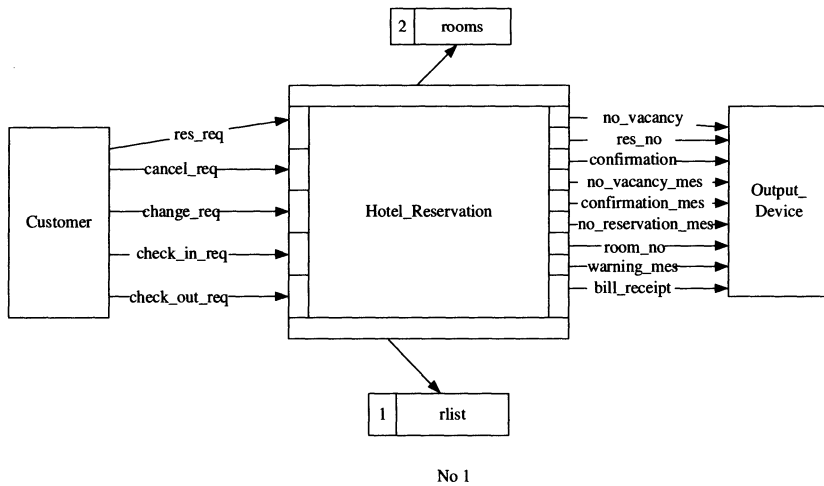
Fig. 14.4. The top level CDFD of the Hotel Reservation System

ext rw rlist
     rw rooms
post (**bound**(res_req) => **bound**(no_vacancy) **or bound**(res_no))
     and
     (**bound**(cancel_req) => **bound**(confirmation))
     and
     (**bound**(change_req) => **bound**(no_vacancy_mes) **or**
                        bound(confirmation_mes))
     and
     (**bound**(check_in_req) => **bound**(no_reservation_mes) **or**
                       bound(room_no))
     and
     (**bound**(check_out_req) => **bound**(warning_mes) **or**
                     bound(bill_receipt))
comment
This process specifies only the relation between the input data flows and
output data flows. In other words, it specifies only which input data flows
are consumed to produce which output data flows. The details of how the
input data flows are used to produce the output data flows are spelled out
in its decomposition.
end_process;
end_module;

The top level CDFD, given in Figure 14.4, involves the terminators Customer, providing requests, and Output_Device, displaying or printing out re-

sults; a process Hotel_Reservation; and two existing external stores rlist and
rooms. Since these two stores are intended to represent independent entities
of this reservation system, we use sharp mark # to indicate this feature of the
store variables when they are declared in the module, as we have explained
in Section 4.13 of Chapter 4. The defined types, variables, and invariants
are derived from the semi-formal specification, with necessary extensions and
modifications. The process Hotel_Reservation is an abstraction of the entire
system; it is intended to describe the overall functionality of the system, with-
out giving details of how the functionality is actually realized; the details are
expected to be spelled out in its decomposition.

   According to the postcondition of process Hotel_Reservation, a reserva-
tion request res_req results in the generation of either output data flow
no_vacancy or res_no; a cancellation request cancel_req leads to the gen-
eration of confirmation; a change reservation request change_req results in
the generation of either no_vacancy_mes or confirmation_mes; a check-in re-
quest check_in_req leads to the generation of either no_reservation_mes or
room_no; and a check_out_req results in the generation of warning_mes or
bill_receipt.

   The process Hotel_Reservation is decomposed into the CDFD in Figure
14.5, which is associated with the module Hotel_Reservation_Decom. This
module is a formalization of the corresponding module in the semi-formal
specification. For brevity, we present only the interesting parts in detail, and
give an outline of the other parts.

   module Hotel_Reservation_Decom / SYSTEM_Hotel_Reservation;

   type

   Date = **nat0** * **nat0** * **nat0**;
   ReservationRequest = **composed of**
                         name: FullName
                         address: **string**
                         tel: **nat0**
                         period: Date * Date
                         room_type: {<Single>, <Twin>, <Double>}
                         end;
   CancellationRequest = **composed of**
                         reservation_no: **nat0**
                         name: Customer.FullName
                         end;
   ChangeRequest = **composed of**
                         reservation_no: **nat0**
                         name: Customer.FullName
                         room_type: {<Single>, <Twin>, <Double>}
                         end;
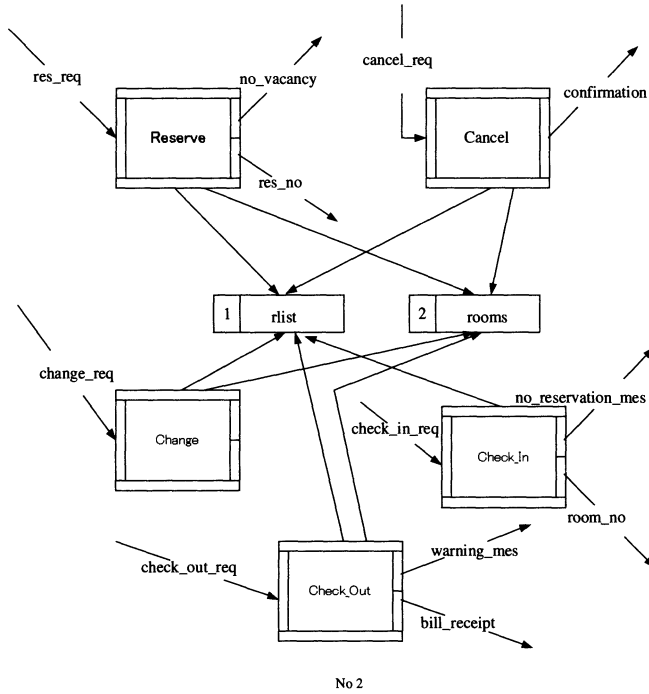
Fig. 14.5. The decomposition of the process Hotel_Reservation

var

ext #rlist: ReservationList;
ext #rooms: Rooms;

behav CDFD_No2;

process Reserve(res_req: ReservationResquest)
                 no_vacancy: **string** | res_no: **nat**
ext wr rlist
     wr rooms
pre not exists[c: dom(rlist)] |
          c.name = res_req.name **and**
          c.address = res_req.address
decom Reserve_Decom
comment

This process reserves a room according to the request,
if there is vacancy. Since this process has a decomposition,
the postcondition is given as **true**, which is omitted in the
specification.
end_process;

process Cancel(cancel_req: CancelRequest)
                confirmation: **string**
ext wr rlist
     wr rooms
pre exists[c: **dom**(rlist)] |
          c.reservation_no = cancel_req.reservation_no **and**
          c.name = cancel_req.reservation_no
...
end_process;

process Change(change_req: ChangeRequest)
               no_vacancy_mes: **string** |
                confirmation_mes: **string**
ext wr rlist
     wr rooms
pre pre_ Cancel[change_req / cancel_req]
...
end_process;

process Check_In(check_in_req: CheckInRequest)
                no_reservation_mes: **string** |
                 room_no: **nat0**
ext wr rlist
...
end_process;

process Check_out(check_out_req: RoomNo)
                  warning_mes: **string** |
                   bill_receipt: **string**
ext wr rlist
     wr rooms
post (**exists**[c: **dom**(~rlist)] |
          (~rlist(c).room_no = check_out_req **and**
          rlist = **domrb**({c}, ~rlist) **and**
          rooms = **union**(~rooms,
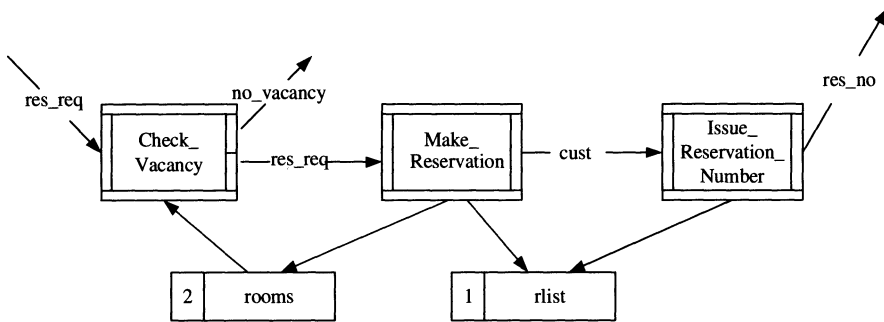
Fig. 14.6. The decomposition of the process Reserve

```
            {modify(~rlist(c), status -> <Check Out>) } and
          bill_receipt = Print_Bill_Receipt(c) ))   or
      warning_mes = "The room number is wrong."
   end_process;


   function Print_Bill_Receipt(c: Customer): string
   == undefined;
   /* this function returns a receipt that may be printed out on a printer. */
   end_function

   end_module;
```

The specification demonstrates a skill in using the SOFL language. For some reason we want to define the precondition of process Change the same as that of process Cancel, except that variable cancel_req needs to be substituted by change_req. This reference is written as **pre_Cancel[change_req / cancel_req]** in the precondition of process Change.

Several processes in the module Hotel_Reservation_Decom are decomposed into lower level modules and their associated CDFDs. As an example, Figure 14.6 shows the decomposition of process Reserve, and the associated module is given as follows:

```
   module Reserve_Decom / Hotel_Reservation_Decom;

   var

   ext rlist: ReservationList;
   ext rooms: Rooms;
```

behav CDFD_No3;

process Check_Vacancy(res_req: ReservationRequest)
                             no_vacancy: **string** |
                             res_req: ReservationRequest
ext rd rooms
post (**exists**[r: rooms] | r.room_type = res_req.room_type) **and**
                     res_req = ~res_req) **or**
    (**not exists**[r: rooms] | r.room_type = res_req.room_type) **and**
    no_vacancy = "No vacancy"
comment
No specific precondition is required. If there exists a room whose type is the same as the required type of the reservation request, pass the reservation request ref_req to the output of this process. Otherwise, produce a "No vacancy" message as its output.
end_process;

process Make_Reservation(res_req: ReservationRequest)
                             cust: Customer
...
end_process;

process Issue_Reservation_Number(cust: Customer)
                             res_no: **nat0**
...
end_process;

end_module;

This module involves three processes: Check_Vacancy, Make_Reservation, and Issue_Reservation_Number. Process Check_Vacancy first checks whether the requested room is available or not. If it is, nothing is done except transferring the reservation request res_req to the next process. Otherwise, a "no vacancy" message is issued. If the available room is confirmed by process Check_Vacancy, process Make_Reservation will reserve a room for the customer, and then activate process Issue_Reservation_Number to issue a reservation number to the customer.

14.4 Evolution

Transformations from informal to semi-formal, and then to formal specifications, are in general an *evolutionary* process. An evolution of a specification can be one of the following three activities:

- Refinement
- Extension
- Modification

Refinement is an activity of improving a specification by resolving non-determinism. The result of a refinement is a concrete specification or program that does exactly what is required in the abstract specification. When a specification (e.g., of a process) is finalized in accordance with the user's agreement, refinement is usually intended to provide a satisfactory program solution. A detailed discussion of refinement is given in Section 14.5.

Extension of a specification means the addition of new components to the specification. A component can be a module, CDFD, process, or even a data type definition. The extension approach emphasizes the reuse of the existing specification in the extended specification; it is therefore appropriate if the existing ideas in the specification need developing. For example, the module Reservation_system and the associated CDFD in the semi-formal specification given in Section 14.2.2 are an extension of the informal specification, and the top level module SYSTEM_Hotel_Reservation in the formal specification defined in Section 14.3 is new to the semi-formal specification.

Modification of a specification is a change, either in syntax or semantics, without conformance to any formalized standard. Since requirements analysis and abstract design involves intensive study of the user's requirements, which may often be changeable in reality, modification is an intrinsic feature of software development. Although modifications are inevitable, it is always desirable to have modifications conducted in a well-controlled manner.

In practice, a combination of refinement, extension, and modification may be employed to develop specifications. The important point is that all of these activities must be performed in a well-controlled manner to ensure the consistency and correctness of all the specifications produced.

14.5 Detailed Design

Detailed design has two goals. One goal is to transform the implicit specifications of processes and functions, defined in modules, into explicit specifications in order for the algorithmic information provided by such explicit specifications to serve as a foundation for implementation in a specific programming language. Another goal is to transform the structured abstract design specification into an object-oriented detailed design specification in order to achieve

good quality of final implementation (e.g., encapsulation, information hiding, reusability, and maintainability). Such a specification will facilitate the implementation using an object-oriented programming language (e.g., Java, C++).

It is worth mentioning that such a transformation needs to keep the hierarchy of CDFDs in the abstract design specification; thus, we can give as much freedom as possible to the programmer in deciding the strategy for the implementation of the specification. For example, the programmer can decide how to implement each CDFD in the specification based on the programming language he or she uses for implementation.

14.5.1 Transformation from Implicit to Explicit Specifications

Since a high level process is defined by its decomposition, there is no need to transform its implicit specification into an explicit one. Only the lowest level processes (i.e., processes with no decompositions) need to be transformed from the implicit specification into an explicit one. The transformation from an implicit specification into an explicit specification is in fact a functional *refinement*. Refinement is an activity of improving a specification by resolving non-determinism. In other words, a refined specification must do whatever is required by the abstract specification, but can make a choice in resolving non-determinism. Note that during a transformation from an implicit specification to an explicit specification, there may be a need to adjust or modify the definitions of some types given in the abstract design, but data refinement should not be emphasized because this issue will be addressed during the implementation of design specifications. Thus, it can help to avoid additional cost, caused possibly by performing strict data refinement in specifications.

Definition 22. *Let P and Q be two processes. Q is a refinement of P if and only if the following two conditions hold:*

(1) **pre_P => pre_Q**
(2) **pre_P and post_Q => post_P**

Weakening the precondition of P in the refinement allows the refined process Q to have a bigger capacity to deal with more possible inputs, but such a weakening of precondition must be done within the constraint of the second condition on the postcondition: strengthening the postcondition of P in Q. Such a strengthening of postcondition requires that Q provide exactly the same functionality expected by process P. For example, suppose we specify process P as

```
process P()
ext wr x: int
pre x > 0
post x > ˜x
end_process
```

Then, we improve this process into process Q:

```
process Q()
ext wr x: int
pre true
post x = ˜x + 1
end_process
```

Obviously, since both conditions required in Definition 22 are met by processes P and Q, Q is a qualified refinement of P. In this case, we also say that Q *satisfies* P.

The notion of refinement can be easily applied to the refinement of an implicit specification into an explicit specification, as we can treat the implicit process as P and the explicit process as Q. For example, the process P given above is extended by adding an explicit specification:

```
process P()
ext wr x: int
pre x > 0
post x > ˜x
explicit
  x := x + 1
end_process
```

To demonstrate that the explicit specification satisfies the implicit specification, we first need to derive the weakest precondition of the explicit specification based on its structure from the postcondition given in the implicit specification, and then prove that the rules (1) and (2) given in Definition 22 are satisfied by the implicit and explicit specifications. However, this technique is usually difficult to apply in practice due to the requirement for advanced skill and effort in conducting formal proofs.

Another definition of refinement that treats a process either in implicit or explicit style as a relation may be more straightforward in facilitating the application of conventional but practical verification techniques, such as testing and inspection.

Definition 23. *Let P and Q be the implicit and explicit specifications of a process, respectively. Q is an refinement of P if and only if the following condition holds:*

$$\textbf{forall}[\text{x: } \textbf{dom}(P), \text{ y: } \textbf{rng}(Q)] \mid \textbf{pre\_P}(x) \textbf{ and } x \text{ Q } y => \textbf{post\_P}(x, y)$$

where **dom**(P) and **rng**(Q) denote the domain of P and the range of Q, respectively, and x Q y means that x and y have relation Q.

According to this definition, if any value x in the domain of P satisfies the precondition of P, and any value y in the range of Q has relation Q with x, they must satisfy the postcodition of P.

Verification of this refinement obligation can be done by testing, although this is unable to provide a full justification due to the intrinsic limitation of testing. Since our focus in this chapter is on the process of software development using SOFL, the testing techniques are described in detail in Chapter 18.

14.5.2 Transformation from Structured to Object-Oriented Specifications

To transform a structured abstract design specification to an object-oriented detailed design specification, the main task is to build appropriate classes and their relations, if any, by converting appropriate data types in the specification (e.g., composite types) into classes, and to achieve information hiding by converting all the stores in the CDFDs to appropriate objects. Specifically, the following points need to be considered for the transformation:

- Convert a composite type into a class definition in the way that the field variables of the composite type are defined as the attribute variables of the class, and its methods are formed based on the operations on the values of the composite type in process specifications.
- Convert a product type into a class definition in a way similar to that for converting a composite type into a class definition.
- Convert a union type as a class hierarchy in which the union type itself is converted into a superclass and all the constituent types are converted into its subclasses.
- If a store in a CDFD is a value of composite, product, or union type, define an appropriate class and convert the store into an object of the class.
- Create new classes to meet the need for developing the abstract design specification (e.g., developing the function of a process).
- Transform the implicit specification of each process and function into an explicit specification in which objects, if any, are manipulated in the way that the principle of information hiding is not violated (i.e., all the attributes of an object are accessed through its methods). To enhance the robustness of the detailed design specification, the precondition of each process and function in the implicit specification must be taken into account in the explicit specification in a way that a proper measure is taken to deal with the violation of the precondition (e.g., produce an error or warning message if the precondition is not met by the inputs of the process or function).

As an example, we transform the process Check_out in the abstract design given previously into the following explicit specification:

```
process Check_out(check_out_req: RoomNo)
                    warning_mes: string |
                    bill_receipt: string
ext wr rlist
    wr rooms
explicit
 begin
  cus: Customer;

  cus := new Customer;
  cus := get({c: dom(rlist) | (rlist(c).room_no = check_out_req};
  if cus <> nil
  then
    begin
      rooms := union(rooms, {(rlist(cus)).setStatus(<Check Out>)});
      bill_receipt := Print_Bill_Receipt(cus);
      rlist := domrb({res}, rlist)
    end
   else
    warning_mes := "The room number is wrong."
 end
end_process;

class Customer;

type
 FullName = string *  string * string;
  /*first name, middle name, and family name */

 var
  name: FullName
  address: string
  tel: nat0
  pass_no: string
  reservation_no: nat0

 method Init()
  . . .
end_class;

class Room;
```

```
var
 room_no: nat0;
 room_type: {<Single>, <Twin>, <Double>};
 status: {<Reserved>, <Check In>, <Check Out>};

method Init()
 . . .
method setStatus(st: {<Reserved>, <Check In>, <Check Out>})
 explicit
    status := st;
 end_method;
 ...
end_class;
```

The explicit specification of process Check_Out is derived based on the understanding of its implicit specification. Several sub-expressions are reused, but in combination with other expressions. Both Customer and Room are converted into classes, and their values are treated as objects of those classes rather than the values of the corresponding composite types as used in the implicit specification. This leads to an extension of the class Room to include the new method setStatus, which assigns the given value st of the enumeration type to the attribute variable status of an object of the class.

14.6 Program

Program is an implementation of the detailed design in a specific programming language. It is desirable to ensure that a program transformed from a detailed design (an explicit specification) satisfies the specification. However, in comparison with transforming an implicit specification into an explicit specification, this process needs to deal with the refinement of abstract data types defined in the design specification into concrete data types available in the programming language. In general, four levels of transformations are necessary:

1. Transformation of the abstract data types.
2. Transformation of explicit specifications of processes, methods, and functions.
3. Transformation of modules.
4. Transformation of classes.

Transformation from an abstract data type into a concrete data type should be a refinement. That is, all the values defined in the abstract data type must be represented by values of the concrete data type. Thus, the functionality required by a process is allowed to be correctly realized by the program.

Formally, let abs and con denote the abstract and concrete data types, respectively; then, con is a refinement of abs if and only if there exists a *retrieve function*, say Retr, such that:

forall[a: abs] **exists**[c: con] | Retr(c) = a

An explicit specification of a process is usually transformed into a procedure (in Pascal), function (in C), or a method (in Java). This transformation must deal with the transformation of all the built-in operators, predicates, and control statements involved. Since explicit specifications describe deterministic functional requirements for the implementation, the program generated from their specifications must have an equivalent functionality or behavior. That is, given an input, both the explicit specification and the program must produce the same output.

A module can be transformed into a procedure in Pascal or a class in Java. Since SOFL has the feature of object-orientation, Java is considered as the target programming language for the transformation in the following discussion. A module corresponds naturally to a class in Java: all the variables declared in the **var** section of the module are transformed into the instance variables of the corresponding class, and all the processes of the module usually correspond to the methods in the class, possibly with some modifications. In addition, another method needs to be defined in the class to implement the CDFD associated with the module.

A class in the design specification can be transformed into a class in Java almost in the same way as when transforming a module, but there is no need to create a new method in the class of implementation to realize the CDFD of the class in the design, because the CDFD in a class of the design specification does not play the role of integrating all the methods defined in the same class to form an overall functionality of the class; it is just used as "syntactic sugar" to help illustrate the relation between methods and attribute variables (which are represented by stores in the CDFD) declared in the **var** section of the class for readability. The detailed discussion on transformation from explicit specifications into Java programs is given in Chapter 19.

14.7 Validation and Verification

Validation and verification of specifications are emphasized in the SOFL method to ensure the consistency between the specifications and the user's real requirements and between different level specifications including programs.

Validation of a specification can be done using either specification testing or face-to-face communication based on static analysis of the specification. Its primary purpose is to ensure that the written specification reflects the user's requirements accurately and completely. The benefit of validation of

specifications is to remove faults in the early phase of software development, and to therefore considerably reduce the cost of development.

Verification of a specification aims to ensure that specifications are internally consistent, satisfiable, and really met by their implementations (or programs). An internally consistent specification means that the components and their relations are defined consistently with the syntax and semantics of the SOFL language. For example, a process specification must not violate the invariant of the module in which it is defined. A satisfiable specification ensures the existence of a mathematical model, and therefore an implementation of the system. A specification is met by an implementation if and only if the refinement rules given in both Definition 22 and Definition 23 are satisfied.

Two techniques are provided for the validation and verification of specifications. One is known as *rigorous review*, and another is *testing*. These two techniques can be applied to both high level and low level specifications, implicit specifications and explicit specifications. Their extensions, known as *specification-based rigorous review* and *specification-based testing*, can also help the verification and validation of programs. The detailed introductions to rigorous review and testing for specifications are given in Chapter 17 and Chapter 18, respectively.

14.8 Adapting the Process to Specific Applications

We have suggested a software process model for organizing software development projects, and emphasized its importance in enhancing the reliability and other qualities of final software systems, in previous sections. However, this does not necessarily mean that the process model must always be fully adopted for any kind of software system development. In fact, taking into account the complexity of systems as well as the cost and time needed to develop the systems, the process model can be tailored to properly achieve the best productivity and reliability for specific development projects. For a small-scale system with low complexity, the formal detailed design phase may be omitted, that is, the formal abstract design can be directly transformed into a program, because the abstract specification may be explicit enough for direct implementation. For a software system required to be implemented by a structured programming language, such as Pascal or C, there is no need to transform the structured abstract design into an object-oriented detailed design. Instead, the abstract design may be transformed to a structured detailed design in which more functions may need to be defined to achieve good modularity. For a large-scale system in the familiar application domain (e.g., the developer has experience in developing similar systems before) the explicit object-oriented detailed design specification may be achieved directly from the semi-formal specification, without going through the phase of abstract design, because many existing specification components defined in terms of classes for previous systems may be reused for the design of the present sys-

tem, and process specifications involving invocations of methods of objects are in general more suitable to be expressed using the explicit specification language rather than pre and postconditions.

Although the process model can be tailored to adapt to different applications, the three-step approach to constructing the user requirements and design specifications is desired to apply to almost all kinds of software projects, because it is a natural approach to take and it presents a good balance between comprehensibility in communication with the users and preciseness in designing the systems. It also helps provide a good traceability for system evolution whenever it is necessary, either during development or maintenance.

14.9 Exercises

1. Give an example to explain the difference between evolution and refinement of processes.
2. Construct a formal design specification of *library system* by taking the three steps: informal, semi-formal, and formal specification. The system is required to provide the services: *Borrow*, *Return*, and *Search*. Each of these services should be implemented by a process. The process Borrow registers the data of the borrowed book; Return removes the registered information about the borrowed book; and Search provides the requested information of the wanted book, if it is available.
3. Refine the implicit specifications of all three processes in the library system into explicit specifications.

15

Approaches to Constructing Specifications

In Chapter 14, we have studied the three-step approach to building formal specifications, but the approach only addresses the issue of how to express specifications in different formats; the question of how to create the architecture of specifications from scratch still remains unanswered. In this chapter we focus the discussion on this problem, and introduce two approaches to constructing specifications: *top-down* and *middle-out*. Each approach contains specific strategies for building specifications, and each strategy has its own features and may have different effects on the process of specification construction. The SOFL method does not restrict the use of these approaches, because different approaches can be effective to different problems. Since a pure bottom-up approach usually does not work well in practice, SOFL is not intended to support this kind of approach. For this reason, we do not give any detailed discussion on the bottom-up approach, although the related techniques for synthesizing lower level CDFDs are described when the middle-out approach is introduced in Section 15.2.

15.1 The Top-Down Approach

The top-down approach is a way to build a specification by first constructing the top level CDFD, and then developing it into a hierarchy of CDFDs by repeatedly decomposing processes occurring in some of the CDFDs involved. The top level CDFD presents an abstraction of the entire system, describing the processes, data flows, and stores that are necessary to provide the most interesting information about the system, while the decomposition of processes helps to develop the abstraction into concrete representations.

There are two strategies for building specifications in this approach; they are known as *CDFD-module-first* and *CDFD-hierarchy-first* strategies. The first strategy stresses the importance of the mutual effect of CDFDs and modules in ensuring the quality of the specification, while the second strategy
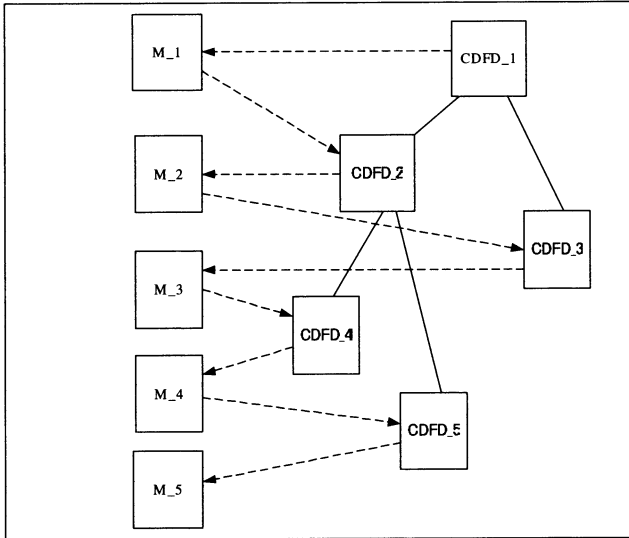
Fig. 15.1. An illustration of CDFD-module-first strategy

emphasizes the importance of the architecture of the system in providing an outline for formalization in the modules.

15.1.1 The CDFD-Module-First Strategy

The fundamental idea of this strategy is that after a CDFD is constructed, its associated module must be defined precisely, before any decomposition of processes in this CDFD takes place. After both the CDFD and module are finalized, the decomposition of another process can take place. Such a process goes on until no process needs further decomposition. Figure 15.1 depicts this strategy. CDFD_1 is the top level CDFD of the specification, and its two processes are decomposed into CDFD_2 and CDFD_3, respectively. Furthermore, two processes of CDFD_2 are decomposed into CDFD_4 and CDFD_5, respectively. Taking the CDFD-module-first strategy, this specification is constructed by first drawing CDFD_1, and then defining module M_1. Then, a process in CDFD_1 is decomposed into CDFD_2, and the associated module M_2 is defined. This process continues until all the CDFDs and their modules are defined.

Since a CDFD usually represents only an outline of an idea, formed on the basis of an initial consideration, it is usually subject to modification when the precise picture of its components and their relations becomes clear. For this reason, before taking any further action in decomposing processes, ensuring desirable components and structure of the current CDFD is important. This can be achieved by defining the associated module of the CDFD. In addition, defining the module may also result in the following effects:

- Improving the understanding of the processes, data flows, and stores involved.
- Improving the structure of the CDFD.
- Identifying processes that need decomposition.

Usually, the meaning and roles of data flows and stores become much clearer when they are defined with specific data types. This is also true of processes when they are specified with pre and postconditions, since nothing can usually be written in the pre and postconditions without a good understanding of the processes. The clarification in formal definitions is likely to help the developer improve the understanding of the components of the current CDFD, and therefore may facilitate modifications of the CDFD. Furthermore, the formalization of the CDFD also serves as a forceful tool for improving the relations among processes, data flows, and stores, as the current relations may be recognized to be incorrect during the formalization.

Determining which processes in a CDFD need decomposition is always a difficult, but important, issue to address. Although it is extremely difficult to give a definitive formula to cope with this problem, the following guidelines may be useful:

1. *If the relation between the input and output data flows of a process cannot be expressed without further information, the decomposition of this process should be considered.*
2. *If the behavior of a process involves a sequence of actions, this process needs to be decomposed.*
3. *If the postcondition of a process is too complex to be written in a concise manner, it may need decomposition.*

The guideline 1 describes a situation in which no relation between the input and output data flows of a process can be defined without further information. This means that some intermediate data flows, which are possibly generated by some intermediate processes, are required to bridge the input and output data flows of the process. The guideline 2 is given because pre and postconditions of a single process are not suitable for defining a sequence of operations. A pair of pre and postconditions can comfortably express only one change of the state, but not many changes. If the postcondition of a process is too complex, it is likely to involve some sort of structure of several operations, so guideline 3 is an *implicit* way to express guideline 2, possibly with some extension.

15.1.2 The CDFD-Hierarchy-First Strategy

Building a specification using the CDFD-hierarchy-first strategy starts with the construction of the CDFD hierarchy by means of decomposition of processes, and then proceeds to define the modules of the CDFDs involved in the

CDFD hierarchy. For example, taking this strategy to build the specification illustrated in Figure 15.1, we will first draw the CDFD hierarchy made up of CDFD_1, CDFD_2, CDFD_3, CDFD_4, and CDFD_5, and then complete their modules M_1, M_2, M_3, M_4, and M_5, respectively.

The CDFD-hierarchy-first strategy has an advantage over the CDFD-module-first strategy: it allows one to create an outline of the entire specification and the foundation for formalization. Also, the formalization can be done with a global view so that the consistency between the interfaces of processes at different levels can be taken into account during the formalization. However, there might be a risk of having to carry out a global modification, when it is found necessary, during the definitions of the modules.

There are two ways to build a CDFD hierarchy. One is the conventional way: *top-down for processes, data flows, and stores*. That is, drawing all the processes, data flows, and stores necessary when creating a new CDFD. However, this can be difficult sometimes, because the data flows and stores necessary at a high level CDFD may not be known precisely. Rather, this information is likely to become clear during the construction of the lower level CDFDs. In this case, another way of building CDFD hierarchies can be applied: *top-down for introducing processes and bottom-up for introducing data flows and stores*. That is, when creating a new CDFD as the decomposition of a high level process, only necessary processes are drawn, without describing the relations among processes in terms of data flows and stores. After the hierarchy of the incomplete CDFDs is formed, the lacking data flows and stores are then added to processes and CDFDs. The addition of data flows and stores usually starts from the lowest level CDFDs and moves up toward the top level CDFD. Of course, there may be changes to be made during the addition of data flows and stores to ensure the structural and semantic consistency between high level processes and their decompositions. Figure 15.2 depicts this approach.

15.1.3 The Modules and Classes

During the building of CDFD and module hierarchies, it is also important to pay attention to defining class hierarchies. However, since classes are primarily treated as user-defined data types, their definitions are attempted whenever the necessity arises during the construction of the CDFD and module hierarchies. Of course, this does not mean we disallow the construction of class hierarchies independently of the CDFD and module hierarchies. If the system under development is within a familiar application domain, building classes as components based on previous experiences can be an effective contribution to the construction of the entire specification. In fact, we can be flexible in using classes and modules in practice, depending on the application domain.

The top-down approach can also be applied to the building of class hierarchies, but based on different notions: generalization and specialization. A high level class is defined to provide common attributes and methods which

(a) Top-dwon for processes

(b) Bottom-up for data flows and stores
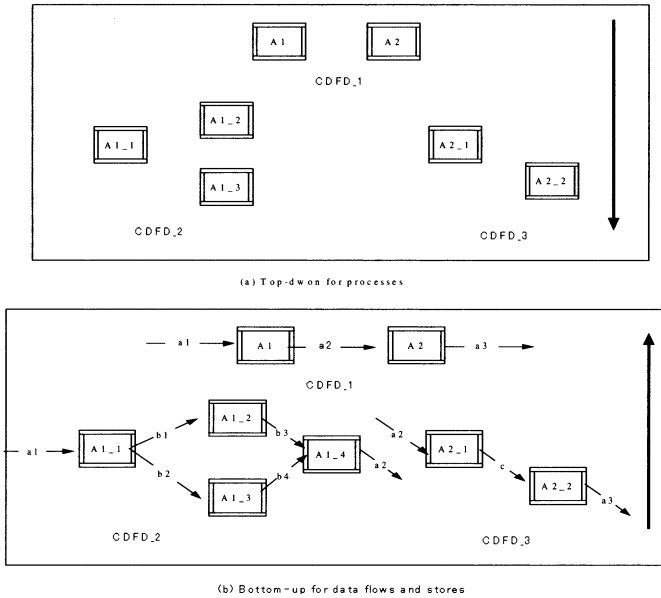
Fig. 15.2. An illustration of the CDFD-hierarchy-first strategy

all its subclasses can inherent. Thus, it represents a generalization of a group of classes. Based on a superclass, subclasses are defined. Such an activity is different from a functional decomposition, as for a process in a CDFD; it actually performs a specialization of its superclass by possibly providing additional attributes and methods.

15.2 The Middle-out Approach

Constructing a specification by the middle-out approach usually starts with the building of the CDFDs and modules modeling the functions that are most familiar to the developer and crucial to the system. These CDFDs and modules are often located somewhere between the top level CDFD and the bottom level CDFDs of the finalized specification hierarchy. On the one hand, for each of these CDFDs the top-down approach is taken to define its processes, until all of the lowest level processes are defined completely and precisely. On the other hand, these CDFDs are used as available components for building high level CDFDs by abstraction and integration. That is, each of these CDFDs is abstracted into a high level process, and then all the high level processes are integrated to form high level CDFDs. Such actions continue until the top level CDFD is reached. Figure 15.3 illustrates this approach. CDFD_2 and CDFD_3 are built first for some reason, and then process A1_3 is decomposed into CDFD_4 to spell out its implementation detail. Finally, CDFD_2 and
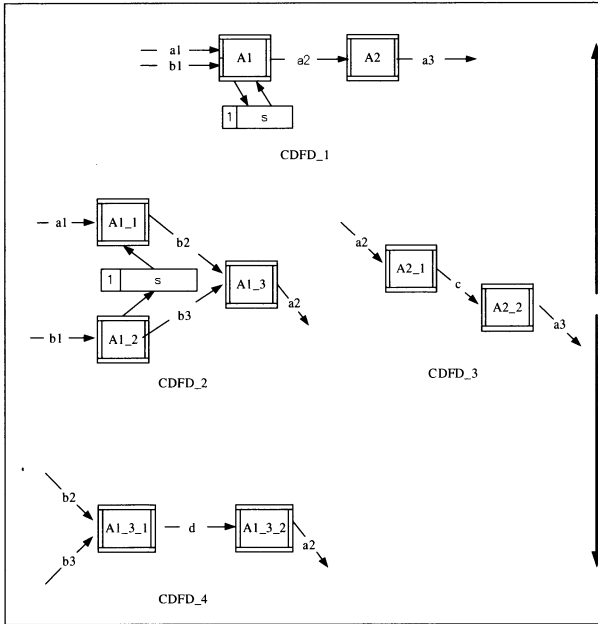
Fig. 15.3. An example of the middle-out approach

CDFD_3 are abstracted into processes A1 and A2 in CDFD_1, respectively, and are integrated into the top level CDFD CDFD_1.

When developing a middle level CDFD by the top-down approach, the same criteria for decomposing processes proposed in Section 15.1.1 can be applied. One of the two strategies, the CDFD-module-first and the CDFD-hierarchy-first strategies, can also be utilized to construct a *local* hierarchy of CDFDs. This hierarchy is seen as local because it would be part of the CDFD hierarchy of the entire specification.

When carrying out integrations of the available CDFDs to build high level CDFDs, the following criteria can be used as guidelines for abstraction:

1. *If there are more than two input data flows to different starting processes of a CDFD, the CDFD needs to be abstracted into a high level process that defines precisely the relationship among those input data flows.* For example, processes A1_1 and A1_2 in CDFD_2 of Figure 15.3 receive data flows a1 and b1, respectively, but at this level the relationship between these two data flows in terms of their availability is unknown. That is, whether both of them are required or only one of them is required to enable the entire CDFD is unknown from this CDFD. Such a relationship between the input data flows can be defined precisely when this CDFD is abstracted into a high level process, such as process A1 in CDFD_1.

2. *If two processes in a CDFD access the same store for both reading and writing, the CDFD needs to be considered for abstraction.* Again, let us look at processes A1_1 and A1_2 in CDFD_2; A1_1 reads from s, whereas A1_2 writes to store s. Since this will cause confusion in accessing and updating store s, processes A1_1 and A1_2 must not be executed concurrently. However, this cannot be ensured at the level of the current CDFD. It is therefore necessary to abstract this CDFD into the high level process A1, which clearly specifies that only one of its input data flows a1 and b1 can be used to enable and execute the process. This restriction will prevent the concurrent executions of processes A1_1 and A1_2 in its decomposition CDFD_2.

3. *If two CDFDs have relations in terms of data flows, they need to be abstracted into high level processes, and the connections between these processes need to be formed in the high level CDFD.* Consider process A1_3 in CDFD_2 and process A2_1 in CDFD_3; the output data flow of A1_3 is the same as the input data flow of A2_1. This indicates that these two processes, belonging to different CDFDs, need communication by data flows. Therefore, the high level processes representing their abstractions should be integrated together in a reasonable form to support the construction of the high level CDFD.

15.3 Comparison of the Approaches

In the previous sections, we have discussed the nature of the top-down and middle-out approaches for building specifications, but have said little about where they can be utilized effectively. Before providing any answer to this problem, we need to get a good understanding of the advantage and weakness of each approach.

The top-down approach is usually effective and intuitive in providing sub-goals or sub-tasks to support the current goal or task, and in developing ideas with little information into ideas with more information. It also provides a good global view of data flows and stores that may be used across CDFDs at different levels; thus the consistency in using data flows and stores can be well managed during the decomposition of high level processes. However, the difficulty in applying this approach may be caused by frequent modifications of high level processes, data flows, stores, and even the CDFDs, as with the progress of decomposition of high level processes. Modifications are necessary because creating accurate components of a high level CDFD in the first place is usually challenging, due to the lack of sufficient knowledge about what data flows and stores will be used or produced by the processes in the lower level CDFDs. To reduce the effect of this problem, the top-down approach for introducing processes and the bottom-up approach for introducing data flows and stores can be helpful.

In contrast to the top-down approach, the middle-out approach may be more effective and natural, because it always starts with modeling the most familiar and crucial functions. It also adopts a flexible way to utilizing the top-down and the bottom-up approaches, and taking the approach which usually stems from natural demands during the construction of the entire specification. However, by using this approach the developer may not find it easy to take a global view of the specification in the early stages; thus data flows, stores, and processes created in different CDFDs may overlap or be defined inconsistently.

Experience suggests that the middle-out approach is effective in requirements analysis and requirements specification constructions, especially for semi-formal cases, because the most familiar and important functional requirements are often focused in the early stages of requirements analysis. While the top-down approach is suitable for design, because the designer usually has a fair understanding of the functional requirements after studying the semi-formal requirements specification, and needs to take a global view in structuring the entire system.

15.4 Exercises

1. Explain the advantages and weaknesses of the top-down and middle-out approaches to building specifications.
2. What is the difference between the CDFD-module-first strategy and the CDFD-hierarchy-first strategy.
3. Build a *Personal Expense Management System* using both top-down and middle-out approaches, respectively. The management system provides the following services: (1) record the expense of an item, (2) search the expenses for a specific item, (3) search for the expense for a kind of item (e.g., cloth, book), (4) update the record of the expense for a specific item, and (5) show the total expense of all the items purchased in a specific month.
4. Rebuild the same *Personal Expense Management System* using both the CDFD-module-first and the CDFD-hierarchy-first approaches, respectively, and compare the advantages and disadvantages of the two different approaches.

A Case Study – Modeling an ATM

The aim of this chapter is to show the entire procedure for developing a formal detailed design specification by evolution and refinement from the informal and semi-formal requirements specifications, and then for formal abstract design specification by describing a systematic case study of modeling an ATM (Automated Teller Machine). It is also intended to show how the structured method can help identify desired functions and then be transformed into an object-oriented detailed design. Although a very simple ATM example is given in Chapter 4, the example is intended to help in the explanation of the module and formal specification of processes, and is not appropriate for showing the entire process of building a formal specification.

Basically, the functional requirements of the ATM are obtained from the informal description of the functionality of the online ATMs of a bank in Japan, but with necessary simplification to suit the purpose of the case study. Even so, the entire contents of the case study are still too large to fit into one chapter of a book. The entire case study contains 69 pages of specifications and is available as a CIS (Faculty of Computer and Information Sciences) technical report of Hosei University [62].

The case study starts from the capturing and documenting of informal requirements, and proceeds to clarify all the operations and data resources involved by the writing of a semi-formal specification. Following the SOFL process model, we then transform the semi-formal specification into a formal abstract design specification to define the architecture and the precise functionality of all the processes and functions involved. Finally, we refine the abstract design specification into a detailed design specification to provide more algorithmic expressions of the functionality of the processes and functions defined in modules in order to facilitate implementation of the system.

By studying this chapter, the reader is expected to deepen his or her understanding of the techniques for the construction of specifications introduced in Chapters 14 and 15, and form a clear picture for the entire process of building a formal specification using SOFL. From the next section, the case study is described in accordance with the SOFL process model step by step.

16.1 Informal User Requirements Specification

The top-down approach is taken to document the informal user requirements specification. To ensure good readability, the specification is organized as a collection of *informal modules* (i.e., informal description of a set of desired operations suitable for being put into a single module). A module is usually composed of potential operations, policy on the operations, and data resources necessary for the operations. Each complex operation in a high level module is decomposed into a low level module, if necessary, and their connection is reflected properly for traceability. Note that since the informal specification is the initial document of the ATM system, the focus is on the potential operations or functions to be provided by the system rather than on the correctness of the syntax of modules. Therefore, the clear shape of modules may not be explicitly seen in the specification. Below is the outline of the informal specification.

    1. The desired functional services: the top-level module:
    (1) Operations on current account.
    (2) Operations on savings account
    (3) Transfer money between accounts
    (4) Manage foreign currency account
    (5) Change password

    2. Decomposition of function (1) in the top-level module
      2.1 Operations

    (1.1) Deposit   /* put money into the current account */
    (1.2) Withdraw   /* get money out of the current account */
    (1.3) Show balance /* display the balance of the current account */
    (1.4) Print out transaction records /* print a list of transactions so far */

     2.2 Policy on operations

    (1) Withdraw: (1.1) Maximum amount to be withdrawn each time is
                1, 000, 000 JPY. /* JPY = Japanese yen */
             (1.2) Maximum amount to be withdrawn each day is
                5, 000, 000 JPY.
             (1.3) No overdraw is allowed.
    (2) Deposit: at most 1, 000,000 JPY can be deposited each time.
    (3) Password is required for all the four operations given above.
    (4) Bank-card is required for all the four operations.
    (5) Bank-book is required only for operation 1.4: print out transaction records.

2.3 Data resources

(1) Each customer has ONE current account.
(2) It is necessary to record the following data items in the system for each customer:
    (2.1) full name
    (2.2) account number
    (2.3) password

3. Decomposition of function (2) in the top-level module
   3.1 Operations

(2.1) Deposit /* put money into the savings account */
(2.2) Application of withdrawing money from the savings account.
        /* withdrawing money from the savings account needs application in advance */
(2.3) Withdraw /*only after a customer submits an application, can he withdraw money from the savings account. */
(2.4) Show balance
(2.5) Print out transaction records

3.2 Data resources

(1) Each customer has ONE savings account.
(2) Each customer needs the following data items to be recorded in the system:
    (2.1) full name
    (2.2) account number
    (2.3) password

3.3 Policy on operations

(1) After every 6 months the customer can withdraw money and money cannot be withdrawn without application in advance.
(2) The maximum amount to be withdrawn each time is 3, 000, 000 JPY. That is, when applying for the withdraw, the customer can apply for up to 3, 000, 000 JPY
(3) The maximum amount to deposit is 3,000,000 JPY

4. Decomposition of function (3) in the top-level module
   4.1 Operations

(1) Transfer money between the current and the savings account
   using cash-card

 4.2 Data resources

(1) The current and savings accounts.

 4.3 Policy on operations

(1) The maximum amount of each transfer transaction is 1,000,000 JPY.

5. Decomposition of function (4) in the top-level module
   5.1 Operations

(1) Purchase US dollars using the money of the current account.
(2) Sell US dollars to JPY and deposit the money into the current account.
(3) Purchase US dollars using cash and deposit the dollars into the foreign
   currency account.
(4) Withdraw JPY from the foreign currency account.
   /*The JPY is converted from US dollars */
(5) Show balance.

 5.2 Data resources

(1) Each customer needs a foreign currency account.
(2) Each customer's following data items need to be recorded in the account:
     (2.1) full name
     (2.2) account number
     (2.3) password

6. Decomposition of function (5) in the top-level module
   6.1 Operations

(1) Change password for the current account.
(2) Change password for the savings account.
(3) Change password for the foreign currency account.

16.2 Semi-formal Functional Specification

The goal of writing the semi-formal specification is to clarify the meaning of all the operations, policies, and data resources that are involved in the informal specification. During this process, undiscovered potential requirements or new aspects of the existing requirements are also expected to be uncovered.

In accordance with the guidelines for transformation from informal specifications to semi-formal ones, given in Section 14.2.2 of Chapter 14, we take the following specific actions to build the semi-formal specification based on the informal one:

- Organize the specification as a set of inter related modules conforming to the SOFL syntax.
- Define all the necessary data types for defining the involved data resources.
- Relate data resources, which are declared as variables of appropriate types, with operations, which are defined as processes, and organize them properly in modules.
- Incorporate the policies on operations into either the pre and postconditions of the corresponding processes or the invariants of the related types and/or state variables.
- Define each process and function (if any) with pre and postconditions, but leave the contents of the pre and postconditions informal.
- When it is necessary, draw a CDFD for a module, but the CDFD may not be a complete one.
- Define composite types in a way the common fields can be reused, that is, try to build a hierarchy of related composite types.

Since the top-down approach allows us to have a global view in defining constants, types, store variables (state variables), and operations, the specific actions described above are taken to construct modules in a top-down manner. However, this does not necessarily mean that the process of building modules has no feedback and change. On the contrary, it involves a lot of changes in the high level modules while the low level modules are being written.

The top-level module, named SYSTEM_ATM, is derived from the top-level module in the informal specification. It declares all the necessary constants, types, and stores for the descendent modules to use, and the necessary processes for functional abstraction. The top-level module is shown below.

```
module SYSTEM_ATM;
const
maximum_withdraw_once = 1,000,000;
/*The unit is JPY, likewise for the following constants.*/
maximum_withdraw_day = 5,000,000;
maximum_deposit_once = 1,000,000;
maximum_withdraw_application = 3,000,000;
ATM_no = i; /*i is any natural number*/
```

type
CustomerInf = **composed of**
              account_no: **nat0**
              pass: Password
              end;
Password = **nat0**; /*A password is a natural number or zero */
AccountInf = **composed of**
              name: **string**   /*The customer's full name */
              balance: **nat0** /*The unit is JPY*/
              transaction_history: **seq of** Transaction
              end;
CurrentAccountInf = AccountInf;
SavingsAccountInf / AccountInf =
          composed of
           withdraw_application_amount: **nat0**
           application_status: **bool** /*true for yes, false for no */
          end;
ForeignCurrencyAccountInf / CustomerInf =
          composed of
           name: **string**
           balance: **real** /*The unit is US dollar */
          end;
CurrentAccountFile = **map** CustomerInf **to** CurrentAccountInf;
SavingsAccountFile = **map** CustomerInf **to** SavingsAccountInf;
ForeignCurrencyAccountFile =
    map CustomerInf **to** ForeignCurrencyAccountInf;
ApplicationNotice = **composed of**
                      application_amount: **nat0**;
                      appication_successful: **bool**;
                      end;
Transaction = **composed of**
                date: Date
                time: Time
                payment: **nat0**
                deposit: **nat0**
                balance: **nat0**
                atm_no: **nat0**
                end;
Date = Day * Month * Year;
Day = **nat0**;
Month = **nat0**;
Year = **nat0**;
var
ext #current_accounts: CurrentAccountFile;
ext #savings_accounts: SavingsAccountFile;

ext #foreign_currency_accounts: ForeignCurrencyAccountFile;
ext #today: Date;
 /*The variable today is assumed to change to reflect
    the date of today in calender.*/
ext #current_time: Time;
 /*This variable represents a clock telling the current time */
inv
forall[x: CustomerInf] | **not exists**[y: CustomerInf] |
   x.account_no = y.account_no;
 /*Each customer's account is unique */
forall[x, y: Transaction] | x <> y;
 /*All the transactions are different. */

process Manage_Current_Account(current: **sign**)
end_process;

process Manage_Savings_Account(savings: **sign**)
end_process;

process Manage_Transfer(transfer: **sign**)
end_process;

process Manage_Foreign_Currency_Account(foreign_currency: **sign**)
end_process;

process Change_Password(change_pass: **sign**)
end_process;

end_module;

Defining data types is one of the most important tasks in writing the semi-formal specification. Since each customer must have a unique account number and password for each kind of bank account, and his or her bank data (e.g., name, balance) must be associated with the customer's account number and password, we declare the composite type CustomerInf for the modeling of the customer's most important information – account number and password – and then define the type AccountInf to represent the information related to the contents of the bank account, including name, balance, and transaction_history. Since each kind of bank account has its own features in addition to the common fields, they are defined as a composite type inheriting from AccountInf, such as SavingsAccountInf and ForeignCurrencyAccountInf.

We need several data stores to represent the collection of the available current accounts, savings accounts, and foreign currency accounts, and these stores need to exist independently of the ATM system (i.e., they should be

available even when the ATM system is not working). For this reason, we declare several existing external stores in the module, such as current_accounts, savings_accounts, and foreign_currency_accounts, each being a map associating the customer's key data (account number and password) with the bank information (e.g., name, balance, and transaction_history) of the corresponding bank account. In addition, we model today and current_time as existing external stores for being used to record the date and time of each bank transaction.

Since each process in the top-level module needs to be decomposed into the next lower level CDFDs, they are defined by specifying both the pre and postconditions as **true** (in this case its pre and postconditions are omitted). We do not draw the CDFD for this module because it is clear enough to reflect the requirements at this level.

The process Manage_Current_Account in the top-module is then decomposed into the module Manage_Current_Account_Decom, as shown below.

```
module Manage_Current_Account_Decom / SYSTEM_ATM;
type
 Notice = composed of
           transaction_account: nat0
           updated_balance: nat0
           end;
var
 ext current_accounts: CurrentAccountFile;

process Current_Authentication(current_inf: CustomerInf)
                       permission: sign | e_mesg1: string
ext rd current_accounts
post if the input account_no and password match those
       of the customer's current account in the store
       current_accounts
     then generate output permission
     else output an error message
end_process;

process Current_Deposit(permission: sign,
                        current_inf: CustomerInf,
                        deposit_amount: nat0)
                         notice: Notice | warning: string
ext wr current_accounts;
post if the input deposit_amount is less than or equal
       to the maximum_deposit_once
     then
       (1) add the deposit_amount to current_account
       (2) give the customer a notice showing the amount
```

of deposit and the updated balance
   (3) update the transaction history of the account
else give a warning message to indicate that the
      amount is over the limit.
end_process;

process Current_Withdraw(permission: **sign**,
                       current_inf: CustomerInf,
                       amount: **nat0**)
                       notice: Notice | warning2: **string**
ext wr current_accounts
post if the input amount is less than or equal to the
      balance of the account and the
     maximum_withdraw_once
   then
    (1) output the cash of the requested amount
    (2) reduce the withdraw amount from the balance
    (3) update the transaction history of the account
    (4) give the notice
   else
     generate the warning message
end_process;

process Current_Show_Balacnce(permission: **sign**,
                          current_inf: CustomerInf)
                          balance: **nat0** | warning3: **string**
ext rd current_accounts
post if the input account_no and pass match those of the
      customer in the store current_accounts
   then display the balance of the customer's current account
   else issue an error message
end_process;

process Current_Print_Transaction_Records(permission: **sign**,
                       current_inf: CustomerInf,
                       date: Date)
                       transaction_records: TransactionRecords
ext rd current_accounts
post print out the transaction records since the input date
end_process;

end_module;

Since all the types declared in the top-level module can be directly used in
the module **Manage_Current_Account_Decom**, only do additional types to be

used in this module, such as the composite type Notice, need to be declared. However, the store variables which are declared in the top-level module but used in the current module, such as current_accounts, are declared as external store variables to indicate the fact that the store variables are used (either be read or updated) in the current module. In the case of variable current_accounts, it is the convention to omit the sharp mark # declaration for an existing external variable when it is not declared in the specification for the first time, so we write **ext** current_accounts rather than **ext** #current_accounts for the declaration of current_accounts in the current module.

Since there is no local store variable, the process Init for initialization of the local store variables is omitted. The other operations listed in the corresponding informal module of the informal specification are defined as processes in the current module with pre and postconditions whose contents are written in an informal manner, such as Current_Deposit, Current_Withdraw, Current_Show_Balance, and Current_Print_Transaction_Records, although their names are slightly different from those in the informal specification. Apart from these processes, we also define a process known as Current_Authentication to ensure security in using the customer's current account. In other words, the functionality of Current_Authentication is to guarantee that only the customer with the correct account number and password can access his current account. As the requirements expressed by the module are quite clear even without its CDFD, it is not given for the module. In fact, there is another important reason why we do not draw the CDFD: the CDFD is usually changed or extended to properly reflect the architecture of design when the corresponding formal abstract design specification is constructed.

Compared with the informal modules, the semi-formal modules are much more precise because all the date structures are well-defined by types and each process is defined by giving a precise signature and reasonably clear description of its functionality through pre and postconditions. This provides a rather firm base for validation against the user requirements and for further formalization in abstract design.

For brevity, we give the outline of the remaining part of the semi-formal specification to help the reader understand the full picture of the specification without going into tedious details.

```
module Manage_Savings_Account_Decom / SYSTEM_ATM;
var
ext savings_accounts: SavingsAccountFile;
    ...    /* process specifications */
end_module;

module Manage_Foreign_Currency_Account / SYSTEM_ATM;
type
ExchangeNotice = composed of
```

                        amount_in_yen: **nat0**
                        current_balance: **nat0**
                        foreign_balance: **real**
                        exchange_rate: **nat0** /*US$1 = n JPY */
                    end;
CashExchangeNotice =
  composed of
    amount_in_yen: **nat0**
    amount_in_dollar: **real**
    foreign_balance: **real**
  end;
var
ext current_accounts: SavingsAccountFile;
ext foreign_currency_accounts: ForeignCurrencyAccountFile;
  ...   /* process specifications */
end_module;

module Change_Password_Decom / SYSTEM_ATM;
var
ext #current_accounts
ext #savings_accounts
ext #foreign_currency_accounts
ext #all_used_passwords: set of Passwords
  ...   /* process specifications */
end_module.

16.3 Formal Abstract Design Specification

There are two main goals of the formal abstract design. One is to define the
system architecture using CDFDs in a hierarchical fashion, and the other is
to formally define the functionality of all the involved processes and functions
by formalizing their pre and postconditions. Through these activities, the
designer is expected to gain a precise understanding of the desired functional
and non-functional requirements, to organize all the necessary processes in the
system architecture in a way that they all work together to provide a solution
for the user requirements, and to build a firm foundation for detailed design
and implementation.

The abstract design specification is constructed by gradually working on
all the modules in the semi-formal specification in a top-down manner. For
example, the formal specification of the top-level module SYSTEM_ATM is
produced based on the corresponding semi-formal module, as shown below.
For brevity, we omit all the constant, type, and store variable declarations, as
well as invariants that are the same as those in the corresponding modules in

the semi-formal specification. Thus, we can focus on the new aspects of the CDFD and the formalization of the modules in the design specification.

```
module SYSTEM_ATM;
 ... /* the same as those in the semi-formal module */
inv
 ... /*the same as those in the semi-formal module */
forall[x, y: {current, savings, transfer, foreign_currency, change_pass}] |
       bound(x) and bound(y) = false;
/*Any two of the input control data flows cannot become
 available at the same time */
behav CDFD_No1;

process Manage_Current_Account(current: sign)
ext wr current_accounts
end_process;

process Manage_Savings_Account(savings: sign)
ext wr savings_accounts
end_process;

process Manage_Transfer(transfer: sign)
ext wr current_accounts
    wr savings_accounts
end_process;

process Manage_Foreign_Currency_Account
                          (foreign_currency: sign)
ext wr foreign_currency_accounts
end_process;

process Change_Password(change_pass: sign)
ext wr all_used_passwords
    wr foreign_currency_accounts
    wr savings_accounts
    wr current_accounts
end_process;

end_module;
```

The CDFD is drawn to represent the design when one of the input data flows of the starting processes (in fact, all the involved processes are starting processes in this particular CDFD) becomes available. This design is reasonable because in reality only one of the bank accounts can be accessed at a time through the same graphical user interface of an ATM. However, since the idea of allowing
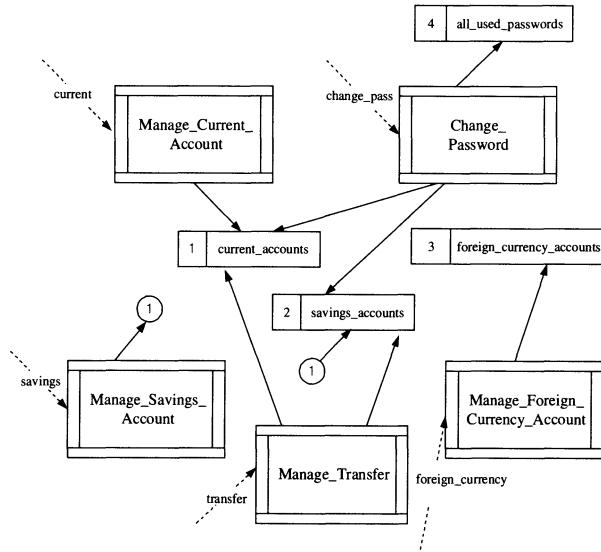
Fig. 16.1. No1

only one of the input data flows to become available at a time is not properly reflected in the CDFD, we define a new invariant, as a universally quantified expression given in the **inv** section, in the module to formalize this idea. The association of the CDFD, which is named **No1**, with the module is reflected by the expression **behav** CDFD_No1. The pre and postconditions of each process in the module is still kept as **true** as they were in the semi-formal module.

The decomposition of process **Manage_Current_Account** is formalized as follows:

> **module** Manage_Current_Account_Decom / SYSTEM_ATM;
> **type**
> ... /* omit the same type declarations */
> OutputDevice = **seq of universal**;
> ServiceCollection = {<1>, <2>, <3>, <4>};
> **var**
> ... /* omit the same variable declarations */
> **ext** #output_device: OutputDevice;
> **inv**
> **forall**[x, y: {deposit, withdraw, s_balance, p_transactions}] |
>   **bound**(x) **and** **bound**(y) = **false**;
> **behav** CDFD_No2;
>
> **process** Select_Services(deposit, a: sign |

                              b, withdraw: sign |
                              c, s_balance: sign |
                              d, p_transactions: sign)
                              sel: ServiceCollection
  post bound(deposit) **and** sel = <1> **or**
      bound(withdraw) **and** sel = <2> **or**
      bound(s_balance) **and** sel = <3> **or**
      bound(p_transactions) **and** sel = <4>
  comment
  The output data flow sel takes different value depending
  on the availability of the input data flows.
  end_process;

  process Current_Authentication(sel: ServiceCollection,
                              current_inf: CustomerInf)
                              current_inf1: CustomerInf |
                              current_inf2: CustomerInf |
                              current_inf3: CustomerInf |
                              current_inf4: CustomerInf |
                              e_mesg1: **string**
  ext rd current_accounts
  post if current_inf **inset dom**(current_accounts)
      then case sel **of**
          <1> -> current_inf1 = current_inf;
          <2> -> current_inf2 = current_inf;
          <3> -> current_inf3 = current_inf;
          <4> -> current_inf4 = current_inf;
          end_case
      else e_mesg1 = "Your password or account number
                          is incorrect."
  comment
   If the input account_no and password match those of
   the customer's current account in the store current_accounts,
  then generate output permission; otherwise, output an error
  message.
  end_process;

  process Current_Deposit(deposit_amount: **nat0**,
                              current_inf1: CustomerInf)
                              notice1: Notice |
                              warning1: **string**
  ext wr current_accounts;
  post if deposit_amount <= maximum_deposit_once
      then
          current_accounts =

override(˜current_accounts,
  {current_inf1 ->
    modify(˜current_accounts(current_inf1),
      balance ->
       ˜current_accounts(current_inf1).balance + deposit_amount,
      transaction_history ->
       conc(˜current_accounts(current_inf1).transaction_history,
        [Get_Transaction(current_accounts, today,
          current_time, 0, deposit_amount, current_inf1)]
)
)
  }
) **and**
  notice1 = **mk_**Notice(deposit_amount,
              current_accounts(current_inf1).balance))
else warning1 = "Your amount is over 1000000 yen limit."
comment
 If the input deposit_amount is less than or equal to the
  maximum_deposit_once,
then
    (1) add the deposit_amount to the current_account
    (2) give the customer a notice showing the amount of
       deposit and the updated balance
    (3) update the transaction history of the account;
else
    give a warning message to indicate that the
amount is over the limit.
end_process;

process Current_Withdraw(current_inf2: CustomerInf,
                     amount: **nat0**)
                     notice2: Notice |
                     warning2: **string**
ext wr current_accounts
post if amount <= maximum_withdraw_one **and**
     amount <= ˜current_accounts(current_inf2).balance
   then
    current_accounts =
     override(˜current_accounts,
        {current_inf2 ->
          modify(˜current_accounts(current_inf2),
            balance ->
             ˜current_accounts(current_inf2).balance
                     - amount,
          transaction_history ->

conc(~current_accounts(current_inf2).transaction_history,
        [Get_Transaction(current_accounts, today,
          current_time, amount, 0, current_inf2)]
)
)
      }
) **and**
      notice2 = **mk_**Notice(amount,
                  current_accounts(current_inf2).balance))
  else warning2 = "Your withdraw amount is over the limit."
comment
If the input amount is less than or equal to the
    balance of the account and the maximum_withdraw_once
 then
      (1) output the cash of the requested amount
      (2) reduce the withdraw amount from the balance
      (3) update the transaction history of the account
      (4) give a notice
else
      generate a warning message
end_process;

process Current_Show_Balacnce(current_inf3: CustomerInf)
                              balance: **nat0**
ext rd current_accounts
post balance = current_accounts(current_inf3).balance
comment
Display the balance of the customer's current account
end_process;

process Current_Print_Transaction_Records(
                  current_inf4: CustomerInf, date: Date)
                  transaction_records: TransactionRecords
ext rd current_accounts
post let transactions =
          current_accounts(current_inf4).transaction_history
      in let i = get({i | i: **inds**(transactions) &
                  transactions(i).date = date})
          in
          transaction_records =
                transactions(i, ..., len(transactions))
comment
Print out the transaction records since the input date
end_process;

process Display_Information(notice1: Notice |
                              notice2: Notice |
                              balance: nat0 |
                              transaction_records:
                               TransactionRecords)
ext wr output_device
post bound(notice1) **and**
    output_device = **conc**(˜output_device, [notice1]) **or**
   bound(notice2) **and**
    output_device = **conc**(˜output_device, [notice2]) **or**
   bound(balance) **and**
    output_device = **conc**(˜output_device, [balance]) **or**
   bound(transaction_records) **and**
   output_device =
      conc(˜output_device, [transactions_records])
comment
Display the input data flows onto the output device
based on their availability.
end_process;

process Display_Message(warning1: string |
                            warning2: string |
                            e_mesg1: string)
ext wr output_device
post bound(warning1) **and**
    output_device = **conc**(˜output_device, [warning1]) **or**
   bound(warning2) **and**
    output_device = **conc**(˜output_device, [warning2]) **or**
   bound(e_mesg1) **and**
    output_device = **conc**(˜output_device, [e_mesg1])
comment
Display the input data flows onto the output device
based on their availability.
end_process;

function Get_Transaction(current_accounts: CurrentAccountFile,
                             to_day: Date,
                             time: Time,
                             pay_amount: nat0,
                             deposit_amount: **nat0**,
                             customer_inf: CustomerInf): Transaction
== **mk_**Transaction(to_day, time, pay_amount, deposit_amount,
               current_accounts(customer_inf).balance,
               ATM_no)
end_function
end_module;
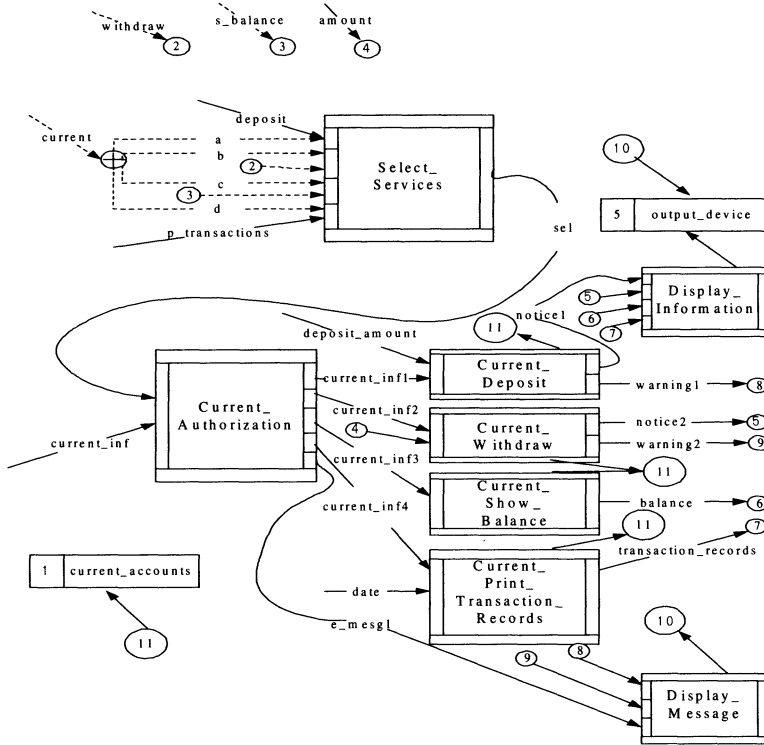
Fig. 16.2. No2

Processes are integrated into the CDFD to define the behavior of the entire module. To ensure that no confusion occurs, which is usually caused by crossing data flows, several connecting nodes are used in the CDFD to indicate the source and destination of the data flows involved. When drawing the CDFD for the module, we realized that some necessary data, such as notices and warning messages, need to be displayed onto an output device. For this reason, we declare the new type OutputDevice and the new existing external variable output_device with this type. Since the output device is expected to accept any type of value produced by the system, we modeled it as a sequence of **universal** (a union type containing values of any types). Apart from the type OutputDevice, another type, ServiceCollection, necessary for modeling the process Select_Services in the CDFD is also defined.

It is practical to allow only one request from the customer to be provided to the ATM at a time; therefore, we define a new invariant to restrict the input control data flows deposit, withdraw, s_balance, and p_transactions from being concurrently available.

Each process in the module is formalized by pre and postconditions, either defining what value is produced for the output data flow variable and/or store variables depending on the availability of the different input data flows, such as the processes Select_Services, Display_Information, and Display_Message, or describing how the output data flows and/or store variables are determined based on the input data flows, such as processes Current_Authentication, Current_Deposit, Current_Withdraw, Current_Show_Balance, and Current_Print_Transaction_Records. Since pre and postconditions of each process is a formalization of the corresponding informal pre and postconditions, the informal ones are reused as comments to interpret the formally defined pre and postconditions in the process specification.

We give for brevity the outline of the remaining part of the entire specification below.

module Manage_Savings_Account_Decom / SYSTEM_ATM;
 ...
end_module;

module Manage_Foreign_Currency_Account / SYSTEM_ATM;
 ...
end_module;

module Change_Password_Decom / SYSTEM_ATM;
 ...
end_module.

16.4 Formal Detailed Design Specification

There are two major tasks in constructing the detailed design specification. One is to transform the structured design specification resulting from the abstract design into an object-oriented design specification by converting and developing all the composite types (as well as product types and union types) involved into classes. Another task is to refine the implicit specification of each process into an explicit specification, providing a more algorithmic expression of the defined behavior of the process. In order to maintain a good traceability of the specification to show the history of building the current version of specification, we always try to keep the existing parts of each process. For example, we still keep the formal expressions of the pre and postconditions for each process while we add the explicit specification to the process definition.

Note that changing the composite types to classes usually does not cause any syntactical change in the declarations of variables because the syntax for declaring a variable with a composite type and a class of the same name have no difference. However, the type declaration part may need updating; for example, the composite type declarations are eliminated because they are

replaced by the corresponding class definitions. Furthermore, in the explicit specifications of the involved processes, operations concerned with the attribute variables of the classes must be implemented by appropriate method invocations.

Again, we take the top-down approach to work out the detailed design specification, as follows:

```
module SYSTEM_ATM;
 ...  /* updated declarations, omitted for brevity */
behav CDFD_No1;

process Manage_Current_Account(current: sign)
ext wr current_accounts
end;

process Manage_Savings_Account(savings: sign)
ext wr savings_accounts
end;

process Manage_Transfer(transfer: sign)
ext wr current_accounts
    wr savings_accounts
end;

process Manage_Foreign_Currency_Account(
                        foreign_currency: sign)
ext wr foreign_currency_accounts
end;

process Change_Password(change_pass: sign)
ext wr all_used_passwords
    wr foreign_currency_accounts
    wr savings_accounts
    wr current_accounts
end;

end_module;
```

The top-level module SYSTEM_ATM is almost unchanged, except for the elimination of the composite types that are converted into the following class definitions:

```
class CustomerInf;
var
 account_no: nat0;
```

 pass: **nat0**;

method Init()
post account_no = 0 **and**
      pass = 0
end_method;
end_class;

class AccountInf;
var
 name: **string**;
 balance: **nat0**;
 transaction_history: **seq of** Transaction;

method Init()
explicit
 begin
  name := ""; /*empty string*/
  balance := 0;
  transaction_history := [ ] /*empty sequence */
 end
end_method;

method Increase_Balance(amount: **nat0**)
ext wr balance
post balance = ˜balance + amount
end_method;

method Decrease_Balance(amount: **nat0**)
ext wr balance
post balance = ˜balance - amount
end_method;

method Update_Transaction_History(transaction: Transaction)
ext wr transaction_history
post transaction_history =
      conc(˜transaction_history, [transaction])
end_method;

end_class;

class CurrentAccountInf / AccountInf;
end_class;

class SavingsAccountInf / AccountInf;

```
var
 withdraw_application_amount: nat0;
 application_status: bool;


method Init()
post withdraw_application_amount = 0 and
     application_status = true
end_method;

method Set_Application_Amount(amount: nat0)
ext wr withdraw_application_amount
    wr application_status
post withdraw_application_amount = amount and
     application_status = true
end_method;

end_class;

class ForeignCurrencyAccountInf / CustomerInf;
var
 name: string;
 balance: real;


method Init()
post name = "" and
     balance = 0.0
end_method;

end_class;

class ApplicationNotice;
var
 application_amount: nat0;
 appication_successful: bool;


method Init()
post application_amount = 0 and
     application_successful = true
end_method;

end_class;
```

```
class Transaction;
type
 CurrentAccountsFile =
          SYSTEM_ATM.CurrentAccountsFile;
var
 date: Date;
 time: Time;
 payment: nat0;
 deposit: nat0;
 balance: nat0;
 atm_no: nat0;

method Init()
explicit
  begin
   date := new Date;
   time := new Time;
   payment = 0;
   deposit = 0;
   atm_no = 0;
 end
end_method;

method Get_Transaction(
                    current_accounts: CurrentAccountsFile,
                    date1: SYSTEM_ATM.Date,
                    time1: SYSTEM_ATM.Time,
                    pay1: nat0,
                    deposit1: nat0,
                    balance1: nat0,
                    current_inf: CustomerInf)
ext wr date
    wr time
    wr payment
    wr deposit
    wr balance
explicit
  begin
   date := date1;
   time := time1;
   payment := pay1;
   deposit := deposit1;
   balance := current_accounts(current_inf).balance;
 end
end  method;
```

```
method Get_Savings_Transaction(
                    savings_accounts: SavingsAccountsFile,
                    date1: SYSTEM_ATM.Date,
                    time1: SYSTEM_ATM.Time,
                    pay1: nat0,
                    deposit1: nat0,
                    balance1: nat0,
                    customer_inf: CustomerInf)
ext wr date
    wr time
    wr payment
    wr deposit
    wr balance
explicit
  begin
   date := date1;
   time := time1;
   payment := pay1;
   deposit := deposit1;
   balance := savings_accounts(customer_inf).balance;
 end
end_method;
end_class;

class Date;
var
 day: nat0;
 month: nat0;
 year: nat0;
method Init()
post day = 0 and
     month = 0 and
     year = 0
end_method;

end_class;

class Notice;
var
 transaction_amount: nat0;
 updated_balance: nat0;
method Init()
post transaction_amount = 0 and
     updated_balance = 0
end_method;
```

```
method Make_Notice(amount: nat0, balance: nat0)
ext wr transaction_amount
    wr updated_balance
explicit
begin
 transaction_amount := amount;
 updated_balance := balance
end
end_method;
end_class;

class TransferNotice;
 var
   transaction_amount: nat0
   from_account_balance: nat0
   to_account_balance: nat0

method Init()
post transaction_amount = 0 and
     from_account_balance = 0 and
     to_account_balance = 0
end_method;

method Make_TransferNotice(transfer_amount1: nat0,
                           from_balance: nat0,
                           to_balance: nat0)
ext wr transaction_amount
    wr from_account_balance
    wr to_account_balance
post transaction_amount = transfer_amount1 and
     from_account_balance = from_balance and
     to_account_balance = to_balance
end_method

end_class;
```

In principle the methods of the classes are defined using implicit specifications if all the state variables (i.e., attribute variables of the related class) involved are basic types (e.g., **nat0**, **int**, and **real**) or compound types built based on them (e.g., **set of nat0**), and using explicit specifications if some of the state variables involved are objects and the invocation of their methods is involved.

Compared with the original composite types, these classes provide more methods to model related operations. The important point is that these methods are derived from the demand in defining processes in CDFDs. In other

words, they are formed when they are necessary for contributing to the building of the entire system. This is much more reasonable than defining classes with imagined or assumed methods from the beginning of a system development.

On the basis of these classes, the module **Manage_Current_Account_Decom** is developed into the specification below to represent the detailed design of the module.

module Manage_Current_Account_Decom / SYSTEM_ATM;
 ... /* the declarations are omitted */
behav CDFD_No2;

process Select_Services(deposit, a: sign |
                                    b, withdraw: sign |
                                    c, s_balance: sign |
                                    d, p_transactions: sign)
                                    sel: ServiceCollection
post bound(deposit) **and** sel = <1> **or**
     bound(withdraw) **and** sel = <2> **or**
     bound(s_balance) **and** sel = <3> **or**
     bound(p_transactions) **and** sel = <4>
explicit
  if bound(deposit)
  then sel := <1>
  else if bound(withdraw)
      then sel := <2>
      else if bound(s_balance)
          then sel := <3>
          else if bound(p_transactions)
              then sel := <4>
comment
The output data flow sel takes different value depending
on the availability of the input data flows.
end_process;

process Current_Authentication(sel: ServiceCollection,
                              current_inf: CustomerInf)
                              current_inf1: CustomerInf |
                              current_inf2: CustomerInf |
                              current_inf3: CustomerInf |
                              current_inf4: CustomerInf |
                              e_mesg1: **string**
ext rd current_accounts
post if current_inf **inset dom**(current_accounts)
     then case sel **of**

```
        <1> -> current_inf1 = current_inf;
        <2> -> current_inf2 = current_inf;
        <3> -> current_inf3 = current_inf;
        <4> -> current_inf4 = current_inf;
        end_case
    else e_mesg1 = "Your password or account
                        number is incorrect."
explicit
    if current_inf inset dom(current_accounts)
    then case sel of
        <1> -> current_inf1 := current_inf;
        <2> -> current_inf2 := current_inf;
        <3> -> current_inf3 := current_inf;
        <4> -> current_inf4 := current_inf;
        end_case
    else e_mesg1 := "Your password or account
                        number is incorrect."
comment
If the input account_no and password match those
 of the customer's current account in the store
 current_accounts
then generate output permission
 else output an error message.
end_process;


process Current_Deposit(deposit_amount: nat0,
                        current_inf1: CustomerInf)
                        notice1: Notice |
                        warning1: string
ext wr current_accounts;
post if deposit_amount <= maximum_deposit_once
    then
      current_accounts =
       override(~current_accounts,
          {current_inf1 ->
            modify(~current_accounts(current_inf1),
              balance ->
              ~current_accounts(current_inf1).balance + deposit_amount,
              transaction_history ->
              conc(~current_accounts(current_inf1).transaction_history,
               [Get_Transaction(current_accounts, today,
                  current_time, 0, deposit_amount, current_inf1)]
                  )
                  )
        }
```

```
          ) and
          notice1 = mk_Notice(deposit_amount,
                           current_accounts(current_inf1).balance))
      else warning1 = "Your amount is over 1000000 yen limit."
explicit
account_inf: CurrentAccountInf;
transaction: Transaction;
begin
    account_inf := new CurrentAccountInf;
    transaction := new Transaction;
    if deposit_amount <= maximum_deposit_once
     then
      begin
       account_inf := current_accounts(current_inf1);
       account_inf.Increase_Balance(deposit_amount);
       account_inf.Update_Transaction_History(
       transaction.Get_Transaction(current_accounts,
                                    today, current_time,
                                    0, deposit_amount,
                                    current_inf1));
       current_accounts :=
         override(current_accounts,
            {current_inf1 -> account_inf});
            notice1 := new Notice;
            notice1.Make_Notice(deposit_amount,
                    current_accounts(current_inf1).balance)
       end
      else warning1 := "Your amount is over 1000000 yen limit."
    end
comment
 If the input deposit_amount is less than or equal to the
   maximum_deposit_once
 then
      (1) add the deposit_amount to the current_account
      (2) give the customer a notice showing the amount
          of deposit and the updated balance
      (3) update the transaction history of the account
 else give a warning message to indicate that the amount
      is over the limit.
end_process;

process Current_Withdraw(current_inf2: CustomerInf,
                          amount: nat0)
                          notice2: Notice |
                          warning2: string
```

```
ext wr current_accounts
post if amount <= maximum_withdraw_one and
        amount <= ~current_accounts(current_inf2).balance
     then
      current_accounts =
       override(~current_accounts,
        {current_inf2 ->
         modify(~current_accounts(current_inf2),
           balance ->
           ~current_accounts(current_inf2).balance - amount,
           transaction_history ->
           conc(~current_accounts(current_inf2).transaction_history,
              [Get_Transaction(current_accounts, today,
              current_time, amount, 0, current_inf2)]
                )
                )
         }
         ) and
         notice2 = mk_Notice(amount,
                       current_accounts(current_inf2).balance))
     else warning2 = "Your withdraw amount is over the limit."
explicit
account_inf: CurrentAccountInf;
transaction: Transaction;
begin
    account_inf := new CurrentAccountInf;
    transaction := new Transaction;
    if amount <= maximum_withdraw_once and
       amount <= current_accounts(current_inf2).balance
     then
      begin
      account_inf := current_accounts(current_inf2);
      account_inf.Decrease_Balance(amount);
      account_inf.Update_Transaction_History(
      transaction.Get_Transaction(current_accounts,
                                   today, current_time,
                                    amount, 0, current_inf2));
       current_accounts :=
        override(current_accounts,
           {current_inf2 -> account_inf});
      notice2 := new Notice;
      notice2.Make_Notice(amount,
         current_accounts(current_inf2).balance)
      end
     else warning2 := "Your amount is over 1000000
```

yen limit."
  end
comment
If the input amount is less than or equal to the
  balance of the account and the
 maximum_withdraw_once
then
      (1) output the cash of the requested amount
      (2) reduce the withdraw amount from the balance
      (3) update the transaction history of the account
      (4) give a notice
else
      generate a warning message
end_process;

process Current_Show_Balance(current_inf3: CustomerInf)
                                    balance: **nat0**
ext rd current_accounts
post balance = current_accounts(current_inf3).balance
explicit
balance := current_accounts(current_inf3).balance
comment
Display the balance of the customer's current account
end_process;

process Current_Print_Transaction_Records(
                        current_inf4: CustomerInf, date: Date)
                        transaction_records: TransactionRecords
ext rd current_accounts
post let transactions =
        current_accounts(current_inf4).transaction_history
     in let i = get({i | i: **inds**(transactions) &
                      transactions(i).date = date})
         in
          transaction_records =
                transactions(i, ..., len(transactions))
explicit
transactions: **seq of** Transaction;
index: **nat0**;
begin
 transactions :=
      current_accounts(current_inf4).transaction_history;
 index :=
      get({i | i: **inds**(transactions) &
              transactions(i).date = date});

```
 transaction_records :=
      transactions(index, ..., len(transactions))
end
comment
Print out the transaction records since the input date
end_process;

process Display_Information(notice1: Notice |
                                notice2: Notice |
                                balance: nat0 |
                                transaction_records:
                                TransactionRecords)
ext wr output_device
post bound(notice1) and
      output_device = conc(~output_device, [notice1]) or
   bound(notice2) and
      output_device = conc(~output_device, [notice2]) or
   bound(balance) and
      output_device = conc(~output_device, [balance]) or
   bound(transaction_records) and
      output_device =
         conc(~output_device, [transactions_records])
explicit
 if bound(notice1)
 then output_device := conc(output_device, [notice1])
 else if bound(notice2)
     then output_device = conc(output_device, [notice2])
     else if bound(balance)
          then output_device =
               conc(output_device, [balance])
          else output_device =
               conc(output_device, [transactions_records])
comment
Display the input data flows onto the output device
based on their availability.
end_process;

process Display_Message(warning1: string |
                             warning2: string |
                             e_mesg1: string)
ext wr output_device
post bound(warning1) and
      output_device = conc(~output_device, [warning1]) or
   bound(warning2) and
      output_device = conc(~output_device, [warning2]) or
```

bound(e_mesg1) and
  output_device = **conc**(~output_device, [e_mesg1])
explicit
 if bound(warning1)
 then output_device :=
       conc(~output_device, [warning1])
 else if bound(warning2)
    then output_device :=
         conc(output_device, [warning2])
    else output_device :=
         conc(output_device, [e_mesg1])
comment
Display the input data flows onto the output device
based on their availability.
end_process;

end_module;

Writing the explicit specification for each process does not involve data refinement in this case study. The essential job, however, is to find out necessary statements and their correct order for the implementation of the meaning of the implicit specification. The explicit specification is therefore more algorithmic than the implicit one, but this does not imply that the explicit specifications is executable like a program, since quantified expressions and set, sequence, and map comprehensions are likely to have been used. It is extremely difficult, if not impossible, to come up with a general algorithm to automatically implement those complicated expressions. Nevertheless, the explicit specification definitely narrows the gap between the implicit specification and its potential program.

16.5 Summary

This case study has demonstrated that SOFL is effective and helpful in three aspects. One aspect is the effectiveness of the combination of CDFDs and formal definitions of their components in the modules: CDFDs provide graphical views of the architecture of the modules, while the formalization of their components help to precisely define the components and improve the structure of the CDFDs. Another important point is the good traceability due to the systematic documentation mechanism in the specifications. This point is especially useful for specification modification. The final point is the reuse of the high level specifications in the low level specifications; thus, the gradual way of developing the formal specification is not a waste of time and effort, but creates necessary documents for progress toward the final formal specification. For example, all the data declarations (e.g., types, store variables, invariants)

are almost copied to the formal specification, and the informal descriptions of
the pre and postconditions of processes are used as comments in their formal
specifications.

On the other hand, drawing and updating CDFDs may take time and
need great care for internal consistency. However, this problem can be ad-
dressed by a powerful software supporting tool. The issues of software tools
and environments are discussed in Chapter 20.

16.6 Exercises

1. Give a semi-formal specification for the module
    Manage_Savings_Account_Decom.
2. Give a formal abstract design specification for the module
    Manage_Savings_Account_Decom.
3. Write a formal detailed design specification for the module
    Manage_Savings_Account_Decom.

17

Rigorous Review

In order that a specification serve as a trustable contract between the developer and the user, and a firm foundation for implementation, it must be ensured that the specification contains no faults or as few faults as possible. Many studies have shown that detecting faults in specifications help substantially reduce the cost and risk of software projects [10]. In this chapter, we introduce a technique known as *rigorous review* for verifying and validating specifications.

Review is a traditional technique for static analysis of software to detect faults that undermine its reliability [28]. Basically, software review means to *check through* software in an appropriate manner, either by a team or an individual. Since software means both program and its related documentation, such as specification, abstract design, and detailed design, a review can be conducted for every level of documentation. Various review methods have been proposed and/or applied in practice with different names, such as *software reviews*, *walk-through*, *static analysis*, and *code inspection* [95][33][114][60].

When dealing with specifications with no formal semantics, the review techniques have to be applied intellectually, based on reviewers' experience, and may not be supported systematically in depth. However, for formal specifications, more rigorous review techniques can be applied. To make reviews effective, especially for complex systems, it is important to use a systematic method that allows the reviewer to focus on a manageable component at each time, and provides an automatic analysis based on the review results of all the related components. The rigorous review introduced in this chapter is a technique developed along this line.

17.1 The Principle of Rigorous Review

The fundamental idea of rigorous review is to improve the rigor and comprehensibility of reviews by utilizing the advantages of formal proof and traditional review techniques as well as appropriate graphical notations. All the

important properties of a specification are expressed as predicate expressions, and conventional review techniques are used to check the properties. A rigorous review is conducted by following the three steps. First, derive properties to be reviewed. Second, for each property generate a graphical representation analyzing the property. Finally, review all the necessary components occurring in the graphical representation. The conclusion of the review for each property is given based on the review results of all the components.

In order that a specification accuratly reflect the user requirements and guarantee a program solution, it is essential to ensure the *internal consistency*, *satisfiability*, and *validity* of the specification.

Definition 24. *A specification is said to be internally consistent if and only if there is no contradiction with the semantics of the SOFL language in the specification.*

Specifically, the internal consistency of a SOFL specification is divided further into several aspects: *internal consistency of process*, *invariant-conformance consistency*, and *internal consistency of CDFD*, each being discussed in detail in Sections 17.2.1, 17.2.2, and 17.2.4, respectively.

Definition 25. *A specification is satisfiable if and only if there exists a mathematical model representing the semantics of the specification.*

Such a model ensures the existence of an implementation for the specification.

Definition 26. *A specification is valid if and only if it satisfies the user's requirements.*

A property is usually represented by a predicate expression, possibly a quantified predicate expression. It can usually be derived from the formal specification, based only on the syntactic structure of the specification.

The graphical notation used in the rigorous review technique for analyzing properties is known as *Review Task Tree*. The detailed description of this notation will be given in Section 17.3. Review task tree notation is a simplified and extended version of the *Fault Tree* notation that is traditionally used for the analysis of safety properties of safety-critical systems whose failure may cause catastrophic disaster to human life and/or important properties [59]. Compared with the fault tree notation, the review task tree notation has several advantages: (1) it takes less space in drawing, (2) it defines review tasks clearly, and (3) it shows the dependency relations among review tasks explicitly.

From the next section, the three steps of rigorous review, mentioned in the beginning of this section, are introduced in detail.

17.2 Properties

Let us focus on a single CDFD and the associated module to discuss, respectively, the four important properties: internal consistency of process, invariant-conformance consistency, satisfiability, and internal consistency of CDFD.

17.2.1 Internal Consistency of a Process

Let M denote a module in which the process P is defined.

> **module** M;
>
> **type**
>
> UsableInt = **int**;
>
> **inv**
>
> I;
>
> **behav** CDFD_1;
>
> **process** P(a: T_1) b: T_2
> **ext wr** x: T_3
>   **rd** y: T_4
>   **wr** z: T_5
> **pre** Q_1
> **post** Q_2
> **end_process**;
>
>
> ...
> **end_module**;

In this module the invariant I and process P are given abstractly, for we intend to discuss the issue of the internal consistency here in general. To express the internal consistency of a process, we need the following notation:

> **Notation**:

- Input(P): the set of all input data flow variables of process P.
- Output(P): the set of all output data flow variables of process P.
- WR(P): the set of all writable (**wr**) external variables (including both decorated and undecorated variables) of process P.
- RD(P): the set of all readable (**rd**) external variables of process P.
- Variables(C): the set of all free variables occurring in condition C.
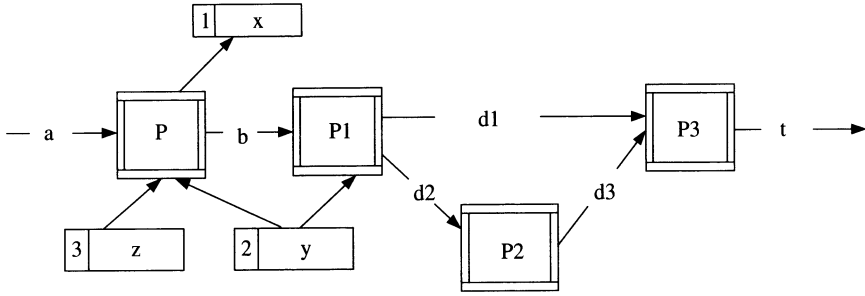
Fig. 17.1. The CDFD of module M

In the definition of internal consistency of a process given below, we assume that the process does not violate the syntactic and typing rules.

Definition 27. *A process* P *is internally consistent if the following conditions hold:*

(1) **forall**[v: Output(P)] | v **notin** Variables(**pre**_P)
(2) v **inset union**(Variables(**pre**_P), Variables(**post**_P)) =>
        v **inset union**(Input(P), Output(P), WR(P), RD(P))

The condition (1) requires that no output variable occur in the precondition of process P, for output variables are only made available as the result of executing the process P. Therefore, the output variables are required to meet the postcondition, but not the precondition. The condition (2) states that any variable used in the pre and postconditions must be one of the input, output, and external variables of the process. In other words, no variables except the input, output, and external variables of the process is allowed to be involved in its pre and postconditions.

For example, let process P be specialized into the following specific process:

```
process P(a: UsableInt) b: UsableInt
ext wr x: UsableInt
    rd y: UsableInt
    wr z: UsableInt
pre  a > 0 and y > 0
post x = a + y and b > x - a and z = ~z + a
end_process;
```

Then,

Input(P) = {a}

Output(P) = {b}
WR(P) = {x, z, ˜x, ˜z}
RD(P) = {y}
Variables(**pre**_P) = {a, y}
Variables(**post**_P) = {x, a, y, b, z, ˜z}

This process is internally consistent, according to Definition 27.

17.2.2 Invariant-Conformance Consistency

The invariant-conformance consistency means that any invariant defined in a module must not be violated by the pre and postconditions of any process defined in the specification.

Definition 28. *Let a type invariant I be defined as* **forall** *[x_1: T_1, x_2: T_2, ..., x_n: T_n] | Q(x_1, x_2, ..., x_n) . Then, a process P and invariant I are consistent if and only if the following two conditions hold.*

(1) (**pre**_P(y_1, y_2, ..., y_m) **and**
    (**exists**[x_2: T_2, ..., x_n: T_n] |
      Q(x_1, x_2, ..., x_n)[y_1/x_1]) **and**
    (**exists**[x_1: T_1, x_3: T_3, ..., x_n: T_n] |
      Q(x_1, x_2, x_3, ..., x_n)[y_2/x_2]) **and**
    ... **and**
    (**exists**[x_1: T_1, x_2: T_3, ..., x_n-1: T_n-1] |
      Q(x_1, x_2, x_3, ..., x_n-1, x_n)[y_n/x_n])) <> **false**
(2) (**pre**_P(y_1, y_2, ..., y_m) **and**
    post_P(z_1, z_2, ..., z_w) **and**
    (**exists**[x_2: T_2, ..., x_n: T_n] |
      Q(x_1, x_2, ..., x_n)[y_1/x_1]) **and**
    (**exists**[x_1: T_1, x_3: T_3, ..., x_n: T_n] |
      Q(x_1, x_2, x_3, ..., x_n)[y_2/x_2]) **and**
    ... **and**
    (**exists**[x_1: T_1, x_2: T_3, ..., x_n-1: T_n-1] |
               Q(x_1, x_2, x_3, ..., x_n-1, x_n)[y_n/x_n])) <>

false

where n >= 1; we assume that y_1, y_2, ..., y_m are the variables of types T_1, T_2, ..., T_m (m <= n), respectively; and likewise z_1, z_2, ..., z_w are the variables of T_1, T_2, ..., T_m (w <= n), respectively (assuming that the x_i (i =1..n) are all different from the y_j (j=1..m) and the z_k (k=1..w)).

In other words, invariant I must not be violated by either precondition or postcondition of process P, for the invariant is part of the overall requirements documented in the specification and is required to be sustained throughout the entire system operation. Note that in SOFL an invariant I is regarded as an implicit part of the pre and postcondition of a process. For example, the precondition (or postcondition) of process P discussed above is not only **pre_P(y_1, y_2, ..., y_m)** (or **post_P(z_1, z_2, ..., z_w)**), but is in fact **pre_P(y_1, y_2, ..., y_m) and forall[x_1: T_1, x_2: T_2, x_n: T_n] | Q(x_1, x_2, ..., x_n)** (or **post_P(z_1, z_2, ..., z_w) and forall[x_1: T_1, x_2: T_2, ..., x_n: T_n] | Q(x_1, x_2, ..., x_n)**). Thus, when module M described previously is implemented, it is the programmer's obligation to implement process P in a manner that invariant I is sustained before and after the execution of process P.

Let us take module M given previously as an example to illustrate the conditions for ensuring the consistency between an invariant and a process. Assuming that invariant I given in module M is defined as

forall[i: UsableInt] | i <= 10000

where the type UsableInt contains only integers less than or equal to 10000. Substituting the concrete precondition, postcondition, and the invariant for the corresponding expressions in the conditions (1) and (2) given in Definition 28, the concrete conditions for process P, given previously to be consistent with I, become

1. **((a > 0 and y > 0) and a <= 10000 and y <= 10000) <> false**
2. **((a > 0 and y > 0) and (x = a + y and b > x - a and z = ~z + a) and a <= 10000 and y <= 10000 and x <= 10000 and b <= 10000 and z <= 10000 and ~z <= 10000) <> false**

The review of these conditions to determine whether they hold or not will be discussed in Section 17.3, as an example of rigorous review based on the review task tree analysis. At the moment, let us continue to concentrate on the definitions of properties of interest.

17.2.3 Satisfiability

Before trying to implement a process specification, we must make sure that the specification is satisfiable. Otherwise, the efforts in the implementation may be wasted because there may be no program solution to meet the specification.

Definition 29. *The satisfiability of process P is defined as:*

forall[a, y, ˜x, ˜z: UsableInt] | (**pre_**P(a, y)[˜x/x, ˜z/z] =>
   exists[b, x, z: UsableInt] | **post_**P(a, b, x, ˜x, y, ˜z, z))

The satisfiability requires that, for any input, if the precondition evaluates to **true**, there must exist an output based on which the postcondition evaluates to **true**. Note that an input may be a group of values bound to the corresponding input variables, including input parameters and appropriate external variables. The review of this property for verification will be explained in Section 17.3.

17.2.4 Internal Consistency of CDFD

The internal consistency of a CDFD is an important property necessary for the correctness of the CDFD with respect to its high level process, if we assume that the CDFD is a decomposition of the high level process. Specifically, this concept is defined as follows.

Definition 30. *The internal consistency of a CDFD is a property that the output data flows of the CDFD can be generated based on its input data flows under the condition that the pre and postconditions of all the processes involved in the execution of the CDFD evaluate to true.*

Obviously, a necessary condition for ensuring the internal consistency is that the output data flows of the CDFD are *reachable* from the input data flows (i.e., there exists a path syntactically from the input data flows to the output data flows). In addition, we also need to ensure that each process involved in an execution of the CDFD is consistently defined and the precondition of each process is guaranteed by the operational environment (e.g., the preceding processes). A consistent CDFD ensures that all the output data flows of the CDFD will be produced consistently, but gives no guarantee whatsoever for the correctness of the CDFD with respect to its high level process, because this will depend on whether the CDFD and the high level process satisfy the refinement rules given in Definition 17 of Section 5.4 in Chapter 5. It is very possible that a CDFD is internally consistent, but not correct with respect to its high level process (giving an example is left to the reader as a homework).

Let us consider the CDFD in Figure 17.1 as an example. The overall internal consistency property is expressed as

   bound(a) and pre_P => bound(t) and post_P3

The availability of an **a** that satisfies the precondition of process **P** must lead to the availability of a **t** that satisfies the postcondition of process **P3**. It is not difficult to tell that the output data flow **t** is reachable from the input data flow **a** in Figure 17.1. In addition to the reachability, we also need to ensure that the preconditions of processes **P1**, **P2**, and **P3** are implied by the postconditions of their preceding processes. Formally,

(1) **post_P => pre_P1**
(2) **post_P1 => pre_P2**
(3) **post_P1 and post_P2 => pre_P3**

To facilitate rigorous review of these properties, they are converted into the following equivalent three expressions:

(1') **not post_P or pre_P1**
(2') **not post_P1 or pre_P2**
(3') **not (post_P1 and post_P2) or pre_P3**

17.3 Review Task Tree

17.3.1 Review Task Tree Notation

To review a property described above, there may be several different strategies, and each strategy may indicate how the property, as a top-level task, should be reviewed in order to support a systematic process of rigorous review. Perhaps many notations can be used to represent a review strategy, but such a representation should be comprehensible, and capable of presenting task decompositions to support "team reviews" (in which a review process of a single property may need to be explained to the other team members) and documentation of the review process (which might be required when the final program product is certified by an authorized organization or the customer).

In this section, we present a graphical notation for representing review tasks in a systematic, logical, and hierarchical manner. The notation is known as *review task tree* (RTT). It is derived by simplifying the *fault tree notation* traditionally used for safety analysis of safety-critical systems [59]. Each node of a review task tree represents a *review task*, defining what to do with a property, and it may be connected to "child nodes" in different ways, depending on the type of the node.

Definition 31. *Let S be a specification and P be a property of S. Then, a review task related to P is a property about P.*

There are two kinds of review tasks. One is "the property involved **holds**" and another is "the property involved **can hold**." The former is represented by a rectangle node, while the latter represented by a round-edged rectangle node. Figure 17.2 gives all the possible nodes in each category of the review task tree notation. To help the drawing of large-scale RTT, some nodes for connecting different parts of an RTT are needed. Figure 17.3 gives two connecting nodes used in RTT.

A	B	Property A (or B) **can** hold (or holds) if all of its child properties hold.
A	B	Property A (or B) **can** hold (or holds) if one of its child properties hold.
A	B	Property A (or B) **can** hold (or holds) if all of its child properties hold in the order from left to right.
A	B	Property A (or B) **can** hold (or holds) if one of its child properties holds in the order from left to right
A	B	Property A (or B) **can** hold (or holds) if its only child property holds.
	B	Property B holds if its right child property holds under the assumption that its left child property holds.
A	B	Property A (or B) **can** hold (or holds). It is an atomic property that has no decomposition.

Fig. 17.2. The major components of RTT

1	Input connecting node, meaning the connection will continue to reach the output connecting node with the same number.
1	Output connecting node, meaning it takes the connection from the input connecting node with the same number.

Fig. 17.3. The connecting nodes of RTT

Figure 17.4 shows a simple RTT. It represents that property A can hold if properties B, C, and D hold; property B holds if G or F holds; property C holds if E holds; and property D holds if H can hold and then W holds. In this RTT, the node containing property A presents the overall task for review and the task is decomposed into three sub-tasks that are represented by the three nodes containing properties B, C, and D, respectively. Then, each of the
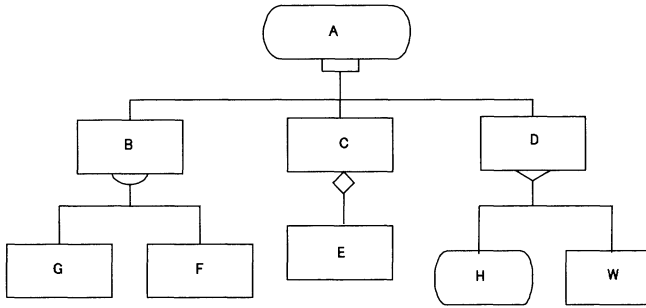
Fig. 17.4. An simple example of RTT

sub-tasks is decomposed further into the smaller tasks: the nodes containing properties G, F, E, H, and W.

Definition 32. *We use* **can_hold** *(P) to represent the task that property P can hold, and* **hold** *(P) to mean that P holds.*

We will often use these two expressions in our discussions concerned with review tasks.

17.3.2 Minimal Cut Sets

Given an RTT, it is important to know what combinations of the leaf tasks, which usually represent the atomic properties for review of the top-level task, will imply the top-level task. Thus, by reviewing appropriate leaf tasks we can check whether the top-level task has been reviewed and whether there exist faults in the related property of the top-level task. For this reason, we need the concept known as *minimal cut sets*.

Definition 33. *Let T be a review task tree. A minimal cut set of T is a smallest combination of the leaf tasks that implies the top-level task of T.*

By the definition, a minimal cut set is thus a combination of the leaf tasks sufficient for implying the top-level task. The combination is a "smallest" one in the sense that reviews of all the tasks in the minimal cut set are needed for performing the review of the top-level task. If one of the tasks in the minimal cut set is not reviewed, then the top-level task cannot be considered to have been reviewed based on this combination. For example, the RTT in Figure 17.4 has two minimal cut sets {G, E, H, W} and {F, E, H, W}, since the combination of each set forms the smallest task implying the top-level task.

For an RTT, its minimal cut sets are finite and unique. Let T denote an RTT, and A its top-level task. Then the minimal cut sets are expressed as follows:

A = M_1 + M_2 + ... + M_n

where M_i (i = 1..n) are minimal cut sets. Each minimal cut set consists of a set of specific leaf tasks, and is expressed as

M_i = {E_1, E_2, ..., E_m}

Thus, the minimal cut set expression for the top-level task of the review task tree given in Figure 17.4 is

A = {G, E, H, W} + {F, E, H, W}

17.3.3 Review Evaluation

The review result of the top-level task of an RTT can be evaluated by means of evaluation of its minimal cut sets. The review result of a task in an RTT has three possibilities: *positive, uncertain,* and *negative.* A positive result means that no fault in the task under review is detected; an uncertain result provides no evidence to either support or deny the task (property); and a negative result indicates that the task contains faults.

Suppose the top-level task A of a review task tree T is expressed in terms of its minimal cut sets

A = M_1 + M_2 + ... + M_n

and each minimal cut M_i (i = 1..n) consists of a set of specific leaf tasks

M_i = {E_1, E_2, ..., E_m}

Then, the way to evaluate the review result of the top-level task based on its minimal cut sets is given through Definitions 34 and 35.

Definition 34. *The review result of M_i (i=1..n) is positive only if the review result of every E_j(j=1..m) is positive, negative if the review result of one of all the E_j(j=1..m) is negative, and uncertain otherwise.*

Definition 35. *The review result of the top-level task A is positive, negative only if the review results of all M_i (i=1..n) are negative, and uncertain otherwise.*

Consider the RTT in Figure 17.4 as an example. Since the top-level task A = {G, E, H, W} + {F, E, H, W}, the review result of A is determined, by definitions 34 and 35, as follows:

- The review result of A is positive if the review results of G, E, H, and W are all positive or the review results of F, E, H, and W are all positive.
- The review result of A is negative if one of the review results of G, E, H, and W is negative and one of the results of F, E, H, and W is negative.
- The review result of A is uncertain if it is neither positive nor negative.

17.4 Property Review

The review of a specification is done by means of reviewing all the important properties discussed in section 17.2 that are derived from the specification. The properties are called *review targets*. To review a property , we take the following steps:

Step 1: Construct a review task tree for the property to show the overall review task and its decomposition.
Step 2: Identify the minimal cut sets of the review task tree.
Step 3: Review all the leaf tasks to determine their truth (remember that a task in a review task tree is also a property).
Step 4: Determine whether the top-level task holds based on the review results of the minimal cut sets.

An RTT for a property (a predicate expression) can be built based on the requirement for and the structure of the property. The requirement for the property forms the top-level task, and it may be decomposed into sub-tasks based on the semantics of its logical expression. A strategy for constructing an RTT for a property is summarized as follows:

- For a compound property (predicate expression), review its constituent predicates first and then its integration.
- For an atomic predicate (e.g., a relation or a negation of an atomic predicate), review whether the set of values constrained by the predicate is empty or not. Such a set must be given in form of set comprehension, so that the types of the involved variables in the predicate will be clearly indicated.

These guidelines serve as a foundation for building review task trees for various kinds of properties, such as those described in Section 17.2.

17.4.1 Review of Consistency Between Process and Invariant

Let us take the process P and the related invariant I given in Section 17.2.1 as an example to show how an RTT can be generated to review the consistency between the process and the invariant. An RTT for reviewing the consistency between the invariant and the precondition of the process is formed based on the review conditions given in Definition 28, as shown in Figure 17.5.
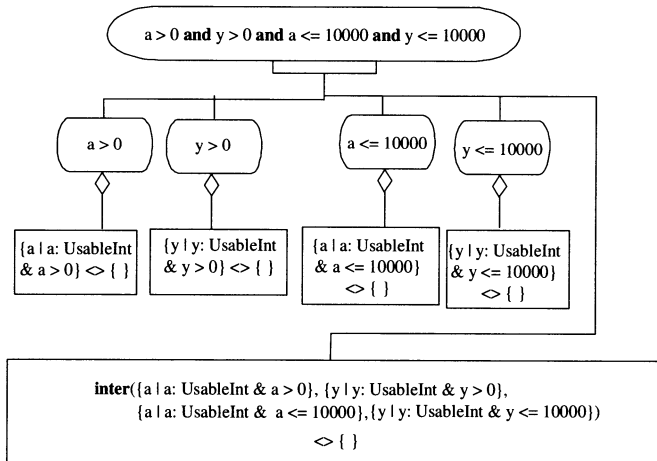
Fig. 17.5. An example RTT for the review of the consistency between the precondition of a process and an invariant

The top-level task of this RTT is **can_hold**(a > 0 **and** y > 0 **and** a <= 10000 **and** y <= 10000), which is equivalent to the property: ((a > 0 **and** y > 0) **and** a <= 10000 **and** y <= 10000) <> **false**. To check whether this top-level task is true, we need to review all of its sub-tasks, such as **can_hold**(a > 0), **can_hold**(y > 0), **can_hold**(a <= 10000), and **can_hold**(y <= 10000). When reviewing a task, say **can_hold**(a > 0), it will be helpful if the reviewer is provided with the type of variable a, because in that way the reviewer can understand not only the involved predicate expression (i.e., a > 0), but also the related context. Furthermore, a straightforward way to show how to determine if **can_hold**(a > 0) is true would be more instructive to the reviewer. To meet these two requirements, we convert the review of the truth of the atomic task **can_hold**(a > 0) into the review of whether the model of the predicate, which is a non-empty set of elements satisfying the predicate (i.e., {a | a: UsableInt & a > 0}), exists or not (i.e., **hold**({a | a: UsableInt & a > 0} <> { })). Obviously, as long as we can find one element of the model, we will be able to assert that **hold**({a | a: UsableInt & a > 0} <> { }) is true. The same principle can also be applied to review the other atomic tasks.

Since only ensuring the truth of each individual sub-task of the top-level task does not necessarily guarantee the truth of the top-level task, there is a need to review whether these sub-tasks are true on the same elements. That is, we need to review whether the task **hold**(**inter**({a | a: UsableInt & a > 0}, {b | b: UsableInt & b > 0}, {a | a: UsableInt & a <= 10000}, {y | y: UsableInt & a <= 10000}) <> { }) is true, as illustrated in the RTT in Figure 17.5.
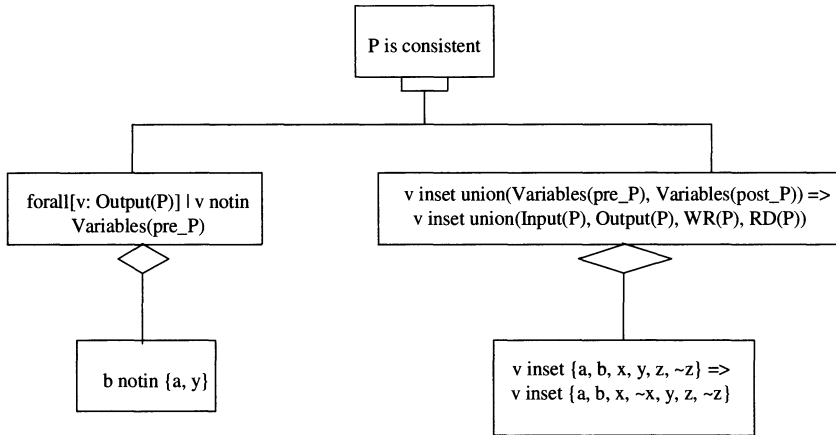
Fig. 17.6. A review task tree for process consistency

It is worth noting that the property in each task in an RTT does not necessarily need to be written in a formal expression, such as a predicate expression; it may be written in an informal language, as long as it facilitates the review of desired properties. We will see more examples of such an RTT below in which the combination of informally and formally described properties are involved.

17.4.2 Process Consistency Review

The consistency of a process specification is defined in Definition 27 in Section 17.2.1. A review task tree for a given process, say P defined in module M in Section 17.2.1, is derived based on the Definition 27 and the concrete specification of P. The review task tree is given in Figure 17.6.

The top-level task of the tree is "P is consistent." There are three intermediate sub-tasks that correspond to the three general conditions given in Definition 27. All the leaf tasks are specialization of their corresponding intermediate tasks by taking the specification of process P into account. For example, the leaf task on the left is a specialization of the intermediate task on the left. Since the only minimal cut set for this review task tree contains all the leaf tasks, if all of them are reviewed and confirmed to be true, then the truth of the top-level task "P is consistent" will be confirmed. In fact, this can be easily concluded by checking the specification of process P. Since the internal consistency of a process is defined solely based on the syntax of the process specification, the derivation of the review task tree and reviews of all the leaf tasks can be performed automatically.

17.4.3 Review of Process Satisfiability

As mentioned in Section 17.2.3, the review target for process satisfiability is the proof obligation given in Definition 29. Let us take the process P specified in Section 17.2.1 as an example to explain the technique for building an RTT. Since the proof obligation for the satisfiability of process P involves an implication in the body of the universally quantified expression, we simplify the proof obligation by converting the body of the universally quantified expression into an equivalent disjunction as follows:

> **forall[a, y, ~z: UsableInt] | not pre_P(a, y, z)[~x/x, ~z/z] or exists[b, x, z: UsableInt] | post_P(a, b, x, ~x, y, ~z, z))**

The RTT built based on this expression is simpler and more comprehensible than that for the original expression of the proof obligation.

We treat the property "P is satisfiable" as the top-level property in the top-level task **hold(P is satisfiable)**, and then decompose the task into subtasks, as shown in Figure 17.7. Note that there is no part in the tree directly corresponding to the universal quantifier, because it is unnecessary to independently check anything related to the type UsableInt for the bindings. The most important part to review is the body of the universally quantified expression. Since it is a disjunction of two constituent expressions, we decompose the top-level task into two sub-tasks, checking **hold(not pre_P(a, y, z)[~x/x, ~z/z])** and **hold(Existentially quantified expression)**. The first task is decomposed into the single task **hold({(a, ~x, ~z) | a, ~x, ~z: UsableInt & not a > 0 and y > 0} <> { })** and the second task is again divided into the two tasks **hold(UsableInt <> { })** and **hold(post_P(a, b, x, ~x, y, ~z, z))**. Finally, the latter task is again decomposed into another single sub-task **hold({(b, x, z) | b, x, z: UsableInt & x = a + y and b > x - a and z = ~z + a } <> { }).**

When reviewing this tree, we just need to review all the atomic tasks in the two minimal cut sets {**hold({(a, ~x, ~z) | a, ~x, ~z: UsableInt & not a > 0 and y > 0} <> { })**} and {**hold(UsableInt <> { }), hold({(b, x, z) | b, x, z: UsableInt & x = a + y and b > x - a and z = ~z + a } <> { })**}, and then evaluate the overall review result based on review results of the two minimal cut sets.

17.4.4 Review of Internal Consistency of CDFD

As described in Section 17.2.4, an internal consistency review for a CDFD aims to check whether an output data flow of the CDFD (which can be an output data flow of a terminating process or node of the CDFD) can be reached from its input data flows (which can be input data flows of some starting processes or nodes of the CDFD) syntactically, and whether the pre and postconditions of all the processes involved in an execution of the CDFD evaluate to true. For example, to review the internal consistency of the CDFD given in Figure
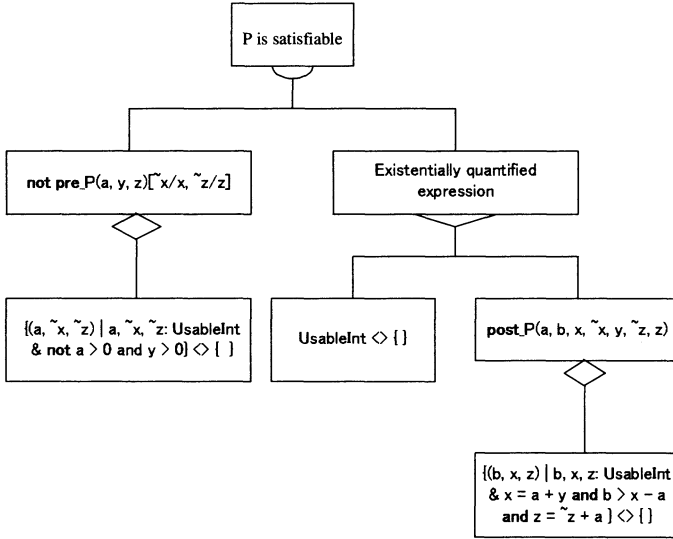
Fig. 17.7. A RTT for reviewing the satisfiability of process P

17.1, it is necessary to check whether the output data flow t can be produced based on the input data flow a through the CDFD. Such checking is equivalent to finding a path of data flows from a to t through the CDFD. However, the existence of such a syntactical path may not necessarily lead to the correct generation of t semantically. For example, when process P1 is not satisfied by its input data flow b, P1 is still executed according to the semantics of CDFD, but its output data flows d1 and d2 may not be correctly produced under the postcondition of P1, which cannot be expected to eventually result in a desired output data flow t.

To ensure the internal consistency of the CDFD, we need to ensure every possible output data flow of the CDFD is consistently produced. In the case of Figure 17.1, we must make sure that the only output data flow t is produced consistently based on the input data flow a through the CDFD. In order to review the consistency of the CDFD, we treat the property **hold(The CDFD of module M is internally consistent)** as the top-level task, and build an RTT as shown in Figure 17.8. The top-level task is ensured by establishing the task **hold(Data flow t is generated consistently)**. According to the semantics of a process, the assurance of the consistent generation of t must be based on the following conditions (tasks):

(1) d1 and d3 are generated consistently.
(2) P3 is satisfiable.
(3) t occurs in the postcondition of P3 (i.e., t **inset** Variables(**post_P3**)).
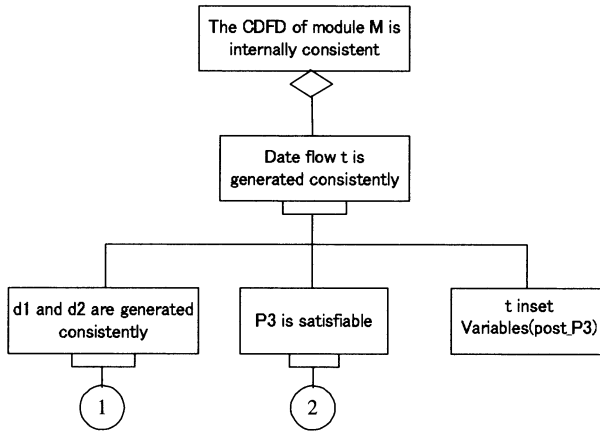
Fig. 17.8. A RTT for the internal consistency of the CDFD of module M

The truth of each of these three conditions may need a further analysis. The consistent generation of data flows d1 and d2 can be analyzed in the same way as that for the analysis of data flow t. The review of the satisfiability of P3 can be done by following the approach described in Section 17.4.3. The truth of condition (3) can be easily found out by a review of the process consistency discussed in Section 17.4.2. For brevity, Figure 17.8 does not give a complete RTT for the review of the internal consistency of the CDFD. All the intermediate tasks whose reviews are already discussed in previous sections in this chapter are connected to an input connecting node, indicating that the existence of another sub-RTT for reviewing the corresponding intermediate task is provided somewhere in the review documentation.

17.5 Constructive and Critical Review

By now we have taken the approach to reviewing properties by trying to establish the properties. That is, suppose we want to ensure property P, we try to establish the sub-tasks that lead to the assurance of P. We call this *constructive review*. However, constructive review is not always possible, due to the lack of necessary information or to the difficulty in doing it explicitly. Since the aim of reviewing a specification is to detect potential faults, one practical way to do it is to review properties by considering the possible reasons for causing them not to hold. This approach is known as *critical review*.

Specifically, suppose we want to ensure property P; we treat **not** P as the top-level property and **hold(not** P**)** as the top-level task of the RTT to review property P. We then develop the review task tree in the same way as that used before for constructive review. In fact, the way of doing review in both constructive and critical approaches are the same: both try to establish the
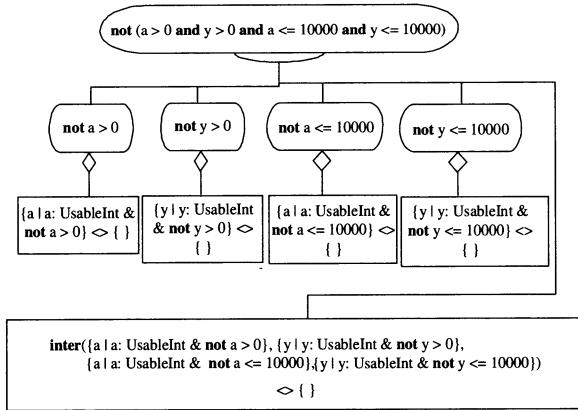
Fig. 17.9. A RTT for critical review of the consistency between an invariant and the precondition of a process

top-level task of a CDFD; the only difference between them is at the content of the top-level property: the top-level property of a constructive review task tree is the property itself, while the top-level property of a critical review task tree is the negation of the top-level property of the corresponding constructive review task tree.

Let us take the RTT in Figure 17.5 as an example to illustrate the critical approach. By treating the negation of the top-level property $a > 0$ **and** $y > 0$ **and** $a <= 10000$ **and** $y <= 10000$ as the top-level property, we build the critical review task tree in Figure 17.9.

17.6 Important Points

The review task tree approach to rigorous review of specifications introduced in this chapter is only a tool for presenting and organizing the review tasks, and for evaluating the review results; it does not conduct reviews automatically. The reviews have to be done by another means, for example, by reading through all the atomic tasks manually, or by testing all the atomic tasks, possibly with some testing tools. While reviews by human reviewers reading through the atomic tasks are usually effective for validating a formal specification, testing can be effective for verifying the consistency properties of the specification. In the next chapter, we will give a detailed introduction to a technique for testing formal specifications.

Although deriving a review task tree for a property of a specification does not ensure that the review of the property will be done satisfactorily, the review task tree approach offers several potential advantages over traditional review techniques:

- the review task tree can be constructed automatically based on a property derived from the specification.
- it allows the reviewer to focus on a manageable review task at a time.
- the review results of manageable tasks can be automatically utilized to determine the result of the overall review.
- the review task tree notation is comprehensible in conveying the ideas of a review, which will be useful and helpful when a review is explained to other people, such as the teammates or the managers involved in the same project.

Since a graphical notation usually occupies more space than texts, skill in drawing review task trees is important. The input and output connecting nodes given in Figure 17.3 are usually very helpful in facilitating the organization of a large review task tree across different pages.

17.7 Exercises

1. Suppose the process P is defined as follows:
   process P(a: **int**) b: **set of int**
   ext wr x: **set of int**
       rd y: **int**
   pre card(x) $<>$ 0
   post inter(x, b) = **union**($\{$a, y$\}$, ˜x)
   end_process
   Build a review task tree for reviewing the internal consistency of process P, and determine whether the process is internally consistent.

2. Build a review task tree for both constructive review and critical review of the satisfiability of process P given above, and determine if process P is satisfiable.

3. Construct review task trees for the "library system" required in Exercise 2 of Chapter 14, to review the following properties: internal consistency of each process involved, the consistency between each process and the invariants (if any), satisfiability of each process, and the internal consistency of all the CDFDs involved in the specification.

18

Specification Testing

Specification testing shares the objective of rigorous reviews, but takes a different approach. It aims to verify various consistency properties of specifications, and to check whether specifications accurately and completely reflect the user requirements by testing the specifications [72][89]. The testing technique introduced in this chapter combines the advantages of formal proof and traditional program testing paradigm, and can be applied to both implicit and explicit specifications. However, since the testing of explicit specifications, which are likely executable, can be done in a way similar to that of testing programs, which have been well researched and studied in the course of software engineering, we focus in this chapter only on the introduction to the technique for testing implicit specifications. Since testing a software system usually requires execution of the system, a special skill for testing implicit specifications, which are usually not executable, is necessary. After studying the process of specification testing, we will go through the details of the testing techniques step by step in this chapter.

18.1 The Process of Testing

Testing a specification consists of three steps. First, generate test cases that may include both *input and output* values for the specification (e.g., process specification). This point may sound strange to the reader who is familiar with the program testing paradigm, in which only input values are required for executing programs, but it is not a mistake. In fact, this point is actually the keypoint that distinguishes the specification testing technique from the traditional program testing paradigm. We will elaborate this point later in Section 18.2.1. Second, evaluate the specification with the test cases, without executing any program implemented based on the specification. Third, analyze test results in order to determine whether faults are detected. This process is illustrated in Figure 18.1.
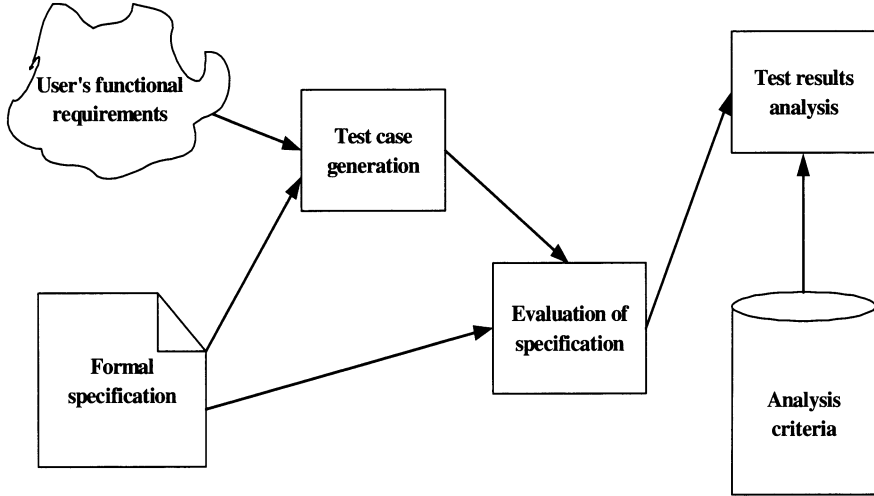
Fig. 18.1. The process of specification testing

The most important task in testing a specification is to create test cases that help uncover as many faults as possible. Two methods for test case generation can be used. One is to generate test cases by examining the structure of the specification itself. This shares the principle of *white-box* testing for programs. Such test cases are usually expected to detect faults leading to the violation of consistency properties of the specification, for example, the satisfiability of a process and the consistency between process specifications and invariants in a module. Several criteria are given in Section 18.3 for generating test cases based on specifications. Another way is to generate test cases based on the user's functional requirements. This method is similar to the *black-box* testing for programs, aiming to validate the specification, that is, to ensure the consistency between the specification and the actual user requirements.

Implicit specifications usually do not indicate algorithms for implementations; therefore, they are not executable in general. However, since they are expressed with predicate expressions, such as pre and postconditions for processes, they can be evaluated if all the variables involved are substituted with concrete values of their types. The results of such evaluations are truth values: **true** or **false**. Of course, they can also be undefined: **nil**. For example, suppose $x > y$ is the precondition of a process; it evaluates to **true** if x is bound to 9 and y bound to 5, and to **false** if x is substituted with 10 and y is substituted with 15.

The analysis of test results is done by comparing the evaluation results with the analysis criteria. Usually, the analysis criteria are predicate expressions, representing the properties to be verified. If the evaluation results are consistent with the predicate expressions, the analysis shows that no faults are detected by the test. Of course, due to the intrinsic limitation of testing (test

cases do not usually cover the entire input domain), this does not necessarily mean that there is no fault existing in the specification.

Testing an entire specification can be conducted at two levels: *unit testing* and *integration testing*. Unit testing aims to detect faults in components, which can be invariants, processes, or methods, in order to ensure their credibility. The credibility of components must be ensured before they are used in the testing of the entire specification, because the credibility of the specification usually relies on that of its components. Integration testing tries to uncover faults occurring in the integration of the components and to check whether the interfaces between components are specified consistently. The testing targets for integration testing are modules, CDFDs, and classes.

18.2 Unit Testing

In this section the testing of processes and invariants are discussed, respectively. To this end, we first need to define the necessary notions.

Definition 36. *Let $P(x\_1, x\_2, ..., x\_n)$ be a predicate expression. A test case for this expression is a group of values $v\_1, v\_2, ..., v\_n$ bound to $x\_1, x\_2, ..., x\_n$, respectively.*

Let us consider the expression $x > y$ **and** $y > z + 1$ as an example. To test its truth, the following three test cases are provided.

(1) $x = 15, y = 8, \ z = 6$
(2) $x = 10, y = 9, z = 5$
(3) $x = 6, y = 10, z = 8$

Definition 37. *A test set for a predicate expression is a set of test cases.*

For example, the three test cases given above form a test set for the predicate expression $x > y$ **and** $y > z + 1$.

Definition 38. *A test suite for a predicate expression is a set of pairs $\{(T\_1, E\_1), (T\_2, E\_2), ..., (T\_n, E\_n)\}$, where each T_i $(i=1..n)$ is a test case and each E_i is an expected result corresponding to the test case.*

As a predicate expression always evaluates to either **true** or **false**, the expected results are actually the truth values. These are different from program testing, where the expected result can be any type of value. This point will become much clearer as our discussion proceeds.

Definition 39. *Let P be a predicate expression and T_s be a test set. Then, a test is a set of evaluations of P with all the test cases in the test set T_s.*

Table 18.1. A test

x	y	E_r	P
-1.0	4	false	false
1.0	0	true	nil
1.5	2	true	true

A test for an expression can be represented by a table. For example, consider the predicate expression P = x > 0 **and** x < 10 / y. A test for this predicate expression is given in Table 18.1, where the column corresponding to E_r shows the expected results and the one corresponding to P presents the actual evaluation results. As we will notice later in this chapter, there is no need of any expected results given explicitly for testing consistency properties. For this reason, the E_r column may not occur in some tests given as examples in this chapter.

18.2.1 Process Testing

The objectives of testing a process are twofold. One is to ensure that the process specification is satisfiable, that is, the specification is possible to implement by a program. Another objective is to ensure that the process specification is valid against the user requirements.

Satisfiability Testing

As discussed in the preceding chapter, a process is satisfiable if its proof obligation can be discharged by formal proof. However, since formal proof may not be cost-effective for large-scale systems in practice, as we mentioned in Chapter 1, we can use testing rather than formal proof to check the proof obligation. Although the proof obligation may not be completely discharged by a test, a rigorous checking of the proof obligation can be performed if the test is carefully designed.

Let us start by looking at a simple process. Suppose process P is defined as follows:

process P(x:**int**) y: **int**
ext wr z: **int**
pre   x > 0 **and** z > 0
post z > x + y + ˜z
end_process

For brevity, this process is represented as

P: [I, O, pre, post]

where I denotes the set of input variables, O denotes the set of output variables, and pre and post are the pre and postconditions of the process, respectively. Specifically, the contents of these components of process P are

> I = {x, ~z}
> O = {y, z}
> pre = x > 0 **and** z > 0
> post = z > x + y + ~z

Note that variable z in the precondition is the same as the variable ~z in the postcondition. To distinguish variable z in the postcondition, denoting the value after the execution of process P, from variable z in the precondition, denoting the value before the execution, we use ~z rather than z in the set of input variables I.

The proof obligation for the satisfiability of process P is:

> **forall**[x, ~z: **int**] |
>   (**pre**_P(x, z)[~z/z] => **exists**[y, z: **int**] |
>   **post**_P(x, ~z, y, z)
>   )

The satisfiability requires that, for any input, if the precondition evaluates to **true**, there must exist an output based on which the postcondition evaluates to **true**. Note that an input may be a group of values bound to the corresponding input variables if there is more than one input variable. Similarly, an output may mean a group of values bound to the corresponding output variables. As we will see later, the proof obligation can be revised to serve as a *test oracle* for test result analysis.

Definition 40. *A test oracle is a logical expression or mechanism that can determine whether a test is successful or not.*

For example, a test oracle for the satisfiability of process P can be derived from the proof obligation given previously by limiting the types of input and output variables to the collections of their values generated in the test

> **forall**[x: T_x, ~z: T_~z] |
>   (**pre**_P(x, z)[~z/z] => **exists**[y: T_y, z: T_z] |
>   **post**_P(x, ~z, y, z)
>   )

where T_x, T_~z, T_y, and T_z denote the set of values generated for variable x, ~z, y, and z in the test, respectively.

To test the satisfiability of a process specification, we need to generate test cases for both input and output variables, because otherwise the evaluation of the postcondition of the process would be impossible. Imagine if we have a program that implements this process specification; the actual output values

Table 18.2. A test for process P

x	~z	z	y	pre	post	pre => post
5	6	20	8	true	true	true
7	8	30	9	true	true	true
-5	9	8	7	false	false	true
8	11	13	7	true	false	false
8	11	40	12	true	true	true

will be produced as a result of the execution of the program with the test cases as input. However, for implicit specifications, such executions are usually impossible.

The primary problem in testing a process specification is how to generate test cases for both input and output variables. We will see several criteria for test case generation in Section 18.3. Until then, let us concentrate on the procedure of testing the satisfiability of the process.

Given test cases for both input and output variables of process P, its pre and postconditions can be evaluated. Table 18.2 shows a test of process P. In this table, pre represents the substituted precondition **pre_P(x, z)[~z/z]** and post the postcondition **post_P(z, ~z, y, z)**. Having this test, we can now analyze the test results to check whether the satisfiability of the process is met or not. In fact, the analysis is simple: we only need to check the results of the implication pre => post for all the test cases. In other words, we can perform this analysis by checking the test oracle for process P. An algorithm for such an analysis derived based on the test oracle is given as follows:

Algorithm

1. If the implication pre => post evaluates to true for all the test cases, the satisfiability of the process is met by the process specification under the current test.
2. If, for any **false** evaluation of the implication pre => post there is no **true** evaluation of the implication based on *the same input values* in the current test, the satisfiability will not hold under the current test. Otherwise, the satisfiability holds under the current test. By **true** or **false** evaluation of a predicate expression we mean that the expression evaluates to **true** or **false**.

The first step of this algorithm is not difficult to understand, because it is consistent with the description of the proof obligation for satisfiability. A little tricky situation is described in the second step. A **false** evaluation of the implication pre => post does not necessarily mean that the implication cannot be met by the same input values, because the result of the evaluation also depends on the output values. For example, the input values x = 8 and ~z = 11, together with the output values z = 13 and y = 7, makes the evaluation of the implication **false**, but makes it **true** when used together with output

Table 18.3. Another test for process P

x	~z	z	y	pre	post	pre => post
5	6	20	8	true	true	true
7	8	30	9	true	true	true
-5	9	8	7	false	false	true
8	11	13	7	true	false	false

values $z = 40$ and $y = 12$. Therefore, despite the **false** evaluation of the implication, the satisfiability of the process still holds under the current test.

However, we must not over-evaluate the credibility of testing in verifying the satisfiability of a process. Even if a test has demonstrated the satisfiability according to the test oracle, this does not necessarily mean that the satisfiability holds for every input of the process, because the test cases in the test oracle usually do not cover the entire domain and range of the process. The only benefit resulting from such a demonstration is to gain confidence in the process specification. This is similar to program testing. By testing we can only establish the existence of faults, but cannot prove the absence of faults, in programs and specifications.

Compared with program testing, process specification testing involves more uncertainties. Even if a test does not meet the proof obligation for the satisfiability of a process, it still does not give sufficient evidence to support the fact that the satisfiability of the process does not hold. Consider the test given in Table 18.2 as an example. If the test case in the bottom row of the table is eliminated, the test will be changed to the one given in Table 18.3. Obviously, under this test we cannot demonstrate that the process is satisfiable because, for the input $x = 8$ and $~z = 11$, there are no output values given in this test that satisfy the postcondition. However, such an output value may exist, but just not be provided in the test.

Definition 41. *A test that does not violate the satisfiability proof obligation of a process is called a* failed test.

Since the objective of testing is to detect faults, well-designed tests must be encouraged. If a test does not detect any fault, it may be bacause of the weakness of the test cases. Of course, there is a possibility of no fault in the specification, but this is usually hard to know. Therefore, considering a test showing no fault as a failed test may encourage more tests to be conducted, which will help improve the confidence in the quality of process specifications. The test given in Table 18.2 is a failed test.

Definition 42. *A test that does not support the satisfiability of a process is known as an* uncertain test.

If a test does not show sufficient evidence to support the satisfiability of a process, like the one given in Table 18.3, it presents an uncertain situation in

verifying the satisfiability of the process: whether the satisfiability holds or not is unknown.

As we will see in discussions throughout this chapter, these two concepts may also apply to the testing of other properties. In general, if a property is shown by a test to hold, the test is said to be a failed test. If a test is unable to either support or deny the property under testing, the test is called an uncertain test. Furthermore, if the property is shown not to hold for some test cases, the test is called a *successful test*.

Definition 43. *If A test uncovers a fault in a property, the test is known as a* successful test *of the property.*

The success of a test is interpreted as a success in detecting faults. Such a connotation may help encourage more successful tests to be conducted, and therefore to improve the quality of specifications.

The results of the discussions above can be easily extended to a more general process that involves two input and output ports:

> **process** Q(x_1: **int** | x_2: **int**) y_1: **int** | y_2: **seq of int**
> **ext wr** z: **int**
> **pre**   x_1 > 0 **and** z > 0 **or** x_2 > 0 **and** z > 0
> **post** z > x_1 + y_1 + ˜z **or** z > x_2 + hd(y_2) + ˜z
> **end_process**

The test cases for testing this process can be generated based on the same criteria, and the test steps are the same. However, one thing is worth noting. Since inputs x_1 and x_2 are exclusive, that is, only one of them is used when the process is executed, we must include the test cases in which one of the two input variables is undefined (i.e., the test value for it is **nil**) when using the other for testing. For example, if x_1 is used for testing process Q, x_2 should be given as **nil**. For output variables y_1 and y_2, the same principle must be applied. Of course, normal values for all the input variables and output variables, even exclusive ones, can be used in tests. Table 18.4 gives a test of process Q, where pre = **pre_Q**(x_1, x_2, z)[˜z/z] and post = **post_Q**(x_1, x_2, y_1, y_2, ˜z, z).

Comparing this result with the test oracle, derived from the proof obligation for the satisfiability of process Q

> **forall**[x_1: T_x_1, x_2: T_x_2, ˜z: T_˜z] |
>   ((**pre_P**(x_1, x_2, z)[˜z/z, **nil**/x_2] =>
>   **exists**[y_1: T_y_1, y_2: T_y_2, z: T_z] |
>   **post_P**(x_1, x_2, y_1, y_2, ˜z, z)[**nil**/x_2, **nil**/y_2])
>   **or**
>   (**pre_P**(x_1, x_2, z)[˜z/z, **nil**/x_1] =>
>   **exists**[y_1: T_y_1, y_2: T_y_2, z: T_z] |
>   **post_P**(x_1, x_2, y_1, y_2, ˜z, z)[**nil**/x_1, **nil**/y_1])
>   )

Table 18.4. A test for process Q

x_1	x_2	y_1	y_2	~z	z	pre	post	pre => post
5	nil	20	nil	4	35	true	true	true
7	nil	30	nil	2	50	true	true	true
-5	nil	8	nil	10	45	false	true	true
nil	11	nil	[2, 5, 8]	5	20	true	true	true
nil	1	nil	[-6, 12]	9	10	true	true	true
nil	3	nil	[9, 2]	-5	20	false	true	true
2	4	1	[9]	3	50	true	true	true

Table 18.5. A test with expected results for process P

x	~z	z	y	pre	Exp_pre	post	Exp_post	pre => post
5	6	20	8	true	true	true	true	true
7	8	30	9	true	true	true	false	true
-5	9	8	7	false	true	false	true	true
8	11	13	7	true	false	false	true	false
8	11	40	12	true	true	true	true	true

we can easily show that process Q is satisfiable under the test given in Table 18.4, because for all the input values satisfying the precondition, the postcondition of the process evaluates to **true**. In the test oracle, T_x_1, T_x_2, $T\_{\sim}z$, T_y_1, T_y_2, and T_z denote the set of all the test values generated for variables x_1, x_2, $\sim z$, y_1, y_2, and z, respectively.

Validity Testing

Since validity testing aims to check whether a process specification is consistent with the user's conception of requirements, the test cases for both input and output variables, as well as the expected test results (truth values), need to be derived from the user's requirements rather than from the process specification itself as we do in satisfiability testing. Apart from this difference, the criterion for analyzing test results is also different. In validity testing, the actual evaluation results of pre and postconditions of a process are compared with the expected results, rather than with the test oracle derived from the proof obligation. Of course, a satisfactory process must also be satisfiable, therefore, satisfiability testing should usually be carried out prior to validity testing. The point is that satisfiability does not necessarily ensure validity with respect to the user's requirements. Let us extend the test given in Table 18.2 for process P to include expected results, which include the expected results of both precondition and postcondition of process P. The extended test is shown in Table 18.5.

In this table, `Exp_pre` and `Exp_post` denote the expected results of precondition and postcondition, respectively. Apparently, the truth value of `Exp_pre` in the fifth row (**false**) is different from the actual evaluation result of the precondition `pre`, and the value of `Exp_post` in the third row (**true**) is different from the actual evaluation result of the postcondition. Although this process specification is demonstrated to be satisfiable by the current test, it does not satisfy the user's requirements because the evaluation results of its pre and postconditions are not exactly the same as the expected results (the expected results are assumed to be correct).

18.2.2 Invariant Testing

An invariant presents a constraint on either types or state variables that must be sustained throughout the entire specification. To ensure this property, two aspects must be checked. One is whether the invariant is feasible, and another is whether the invariant is not violated by all the related processes. For brevity in discussions on the testing method in this section, we focus on invariants involving only a single bound variable in the universally quantified expression. The same method can be extended to invariants containing multiple bound variables.

Feasibility Testing

A feasible invariant of a type must ensure that the type is non-empty.

Definition 44. *Let I denote the invariant **forall** [x: D] | P(x). I is feasible if and only if there exists a value r in D that P(r) holds.*

The fact that type `D` contains some members satisfying property `P` implies that the invariant **forall**[x: D] | P(x) is possible to satisfy with some values. Therefore, requiring that all the members of `D` satisfy the invariant is meaningful, because an empty type is usually neither interesting nor useful in specifications.

To test an invariant for the demonstration of its feasibility, test cases can be generated based on the structure of the invariant, and the following condition

$$\mathbf{exists}[r:\ T\_r]\ |\ P(r)$$

must be used as the test oracle for test results analysis, where `T_r` is a set of values generated for variable `r` in the test, and **subset(T_r, D)** (i.e., `T_r` is a subset of `D`) holds. Let us consider the type `Customer`:

```
Customer = composed of
          id: nat0
          name: string
          end
```

Table 18.6. A test for the feasibility of invariant

x	I
(0, "Mark")	false
(1, "John")	false
(11, "David")	true
(350, "Darrell")	true
(23, "Chris")	true

Assume the type has an invariant

forall[x: Customer] | x.id $>=$ 10 **and** x.id $<$ 1000 **and** **len**(x.name) $<=$ 15

A test oracle derived from this invariant is

exists[r: T_r] | r.id $>=$ 10 **and** r.id $<$ 1000 **and** **len**(r.name) $<=$ 15

where T_r, a subset of Customer, denotes the set of all test values generated for the bound variable x occurring in the invariant.

As an example, a test for this invariant is given in Table 18.6, in which x is a variable over type Customer and I denotes the body of the quantified expression in the invariant x.id $>=$ 10 **and** x.id $<$ 1000 **and** **len**(x.name) $<=$ 15. Apparently, this test is a failed test because it does not detect any fault in the invariant as far as the feasibility is concerned. In fact, in this particular case, the feasibility of the invariant has actually been *proved* by the test, in the sense of providing sufficient evidence to support the truth of the feasibility.

Consistency Testing

As defined in Section 17.2.1 of the preceding chapter, an invariant is consistent with the related process specifications if it is not violated by those processes before and after executions of the processes. Let P be a process and I be an invariant. I is consistent with P if and only if the following two conditions hold:

(1) (**pre_P**(y_1, y_2, ..., y_m) **and**
    (**exists**[x_2: T_2, x_n: T_n] |
        Q(x_1, x_2, ..., x_n)[y_1/x_1]) **and**
    (**exists**[x_1: T_1, x_3: T_3, ..., x_n: T_n] |
        Q(x_1, x_2, x_3, ..., x_n)[y_2/x_2]) **and**
    ... **and**
    (**exists**[x_1: T_1, x_2: T_3, ..., x_n-1: T_n-1] |
        Q(x_1, x_2, x_3, ..., x_n-1, x_n)[y_n/x_n])) $<>$ **false**

(2) (**pre_**P(y_1, y_2, ..., y_m) **and**
$\quad$ **post_**P(z_1, z_2, ..., z_w) **and**
$\quad$ (**exists**[x_2: T_2, x_n: T_n] |
$\quad\quad$ Q(x_1, x_2, ..., x_n)[y_1/x_1]) **and**
$\quad$ (**exists**[x_1: T_1, x_3: T_3, ..., x_n: T_n] |
$\quad\quad$ Q(x_1, x_2, x_3, ..., x_n)[y_2/x_2]) **and**
$\quad$ ... **and**
$\quad$ (**exists**[x_1: T_1, x_2: T_3, ..., x_n-1: T_n-1] |
$\quad\quad$ Q(x_1, x_2, x_3, ..., x_n-1, x_n)[y_n/x_n])) <> **false**

Condition (1) ensures that the precondition of process P does not violate invariant I, while condition (2) ensures that the postcondition does not violate I. Note that if there is no common type or state variable involved in the specification of process P and invariant I, and I is feasible, then conditions (1) and (2) will definitely hold, assuming that the precondition of P is not a contradiction. The reason is obvious: the evaluations of invariant I and the precondition **pre_**P do not interfere with each other.

For example, suppose we define a process Change_Id as

```
process Change_Id()
ext wr cus: Customer
pre   cus.id > 50
post   cus = modify(~cus, id -> 8)
end_process
```

This process changes the id of a customer with identification number greater than 50 to 8. The functionality of this process may not make a good sense, but it helps our discussion at the moment, because the consistency testing is done based entirely on logical expressions, without necessarily considering their meaning.

A test for checking the consistency between the precondition of this process and the invariant I of type Customer described in Section 18.2.2, and between the postcondition of this process and the invariant, is given in Tables 18.7 and 18.8, respectively, where con1 = pre **and** I, con2 = post **and** I, I_body1 denotes the evaluation result of the body of invariant I based on the input ~cus, and I_body2 represents the evaluation result of the body of I based on the output cus.

Since in Table 18.7 the conjunction con1 evaluates to true twice, it provides sufficient evidence to support the claim that the precondition and the invariant are consistent, according to condition (1) defining consistency above. However, the situation in Table 18.8 is very different: the conjunction con2 evaluates to false for all the values of cus. This fact does not support the assertion that the postcondition and the invariant are consistent, but it is also not sufficient to deny this assertion, because the given values for cus are limited. So, the test in Table 18.8 is qualified as an uncertain test. The key point in designing such a

Table 18.7. A test for checking the consistency.

˜cus	pre	l_body1	con1
(0, "Mark")	false	false	false
(11, "David")	false	true	false
(350, "Darrell")	true	true	true
(60, "Chris")	true	true	true

Table 18.8. A test for checking the consistency.

˜cus	cus	pre	post	l_body2	con2
(0, "Mark")	(8, "Mark")	false	true	false	false
(11, "David")	(8, "David")	false	true	false	false
(350, "Darrell")	(8, "Darrell")	true	true	false	false
(60, "Chris")	(20, "Chris")	true	false	true	false

test is to try to provide test cases based on which the pre and postconditions can evaluate to **true**, and so do their conjunctions with the relevant invariants.

18.3 Criteria for Test Case Generation

To test the specification properties discussed so far, we need a rigorous and effective method for generating test cases. Except for the validity testing, test cases are usually generated based on test targets, which are predicate expressions, such as pre and postconditions of a process. The fundamental problem is how test cases should be generated so that they can be used most effectively to detect faults contained in specifications.

One solution to this problem is to provide a set of criteria by which effective test cases can be generated. In this section, we present several criteria, each representing a different testing strategy. The primary idea of these criteria is to provide a framework based on which test cases are generated to meet a desired standard in terms of test coverage.

The discussions of the test criteria are based on the assumption that the predicate expression to be tested is in a disjunctive normal form. Since any predicate expression can be transformed to an equivalent disjunctive normal form by applying deMorgan's laws and the related rules (e.g., distributivity), this assumption is not unreasonable.

Let P = P_1 **or** P_2 **or** ... **or** P_n be a disjunctive normal form and P_i = Q_i_1 **and** Q_i_2 **and** ... **and** Q_i_m be a conjunction of relational expressions Q_i_j (i = 1..n, j = 1..m). Let T_d be a test set (a set of test cases). The test criteria for testing P are described below.

Criterion 1 Evaluate P with T_d to **true** and **false**, respectively.

This criterion is illustrated in Tabel 18.9.

Table 18.9. Criterion 1

P
true
false

Table 18.10. Criterion 2

P_1	P_2	P_3	...	P_n
true	*	*	...	*
false	*	*	...	*
*	true	*	...	*
*	false	*	...	*
*	*	*	...	*
*	*	*	...	true
*	*	*	...	false

A test set meeting this criterion is expected to explore the two situations of predicate P when P evaluates to **true** and **false**, respectively. However, it is worth noting that a tautology is impossible to evaluate to **false**; therefore, this criterion must not be used as a strict measure of the qualification of a test set, but rather as a guideline for the generation of the test set. This principle is also applicable to the other criteria introduced later.

This criterion is easy to apply, and reasonable when testing is only for an abstract level checking, but it is not strong enough for detecting faults because it focuses only on the overall evaluation of the predicate expression: it does not examine every clause of a disjunctive normal form. A stronger criterion is given next.

Criterion 2 Evaluate each P_i (i = 1..n) with T_d to **true** and **false**, respectively.

The idea of this criterion is illustrated in Table 18.10, in which the asterisk * denotes a truth value, either **true** or **false**.

This criterion requires that, by the test set T_d, each disjunctive clause P_i (i = 1..n) evaluates to **true** and **false**, respectively. For example, consider the predicate Q: x - y < 5 **or** x + y > 10; the test of Q given in Table 18.11 satisfies this criterion. Note that this criterion may not give an assurance that each disjunctive clause P_i is tested independently. In other words, a test satisfying this criterion may not allow us to test independently each clause when its evaluation result dominates the evaluation result of the overall predicate expression. Let us take the test in Table 18.11 as an example. The case when clause x + y > 10 evaluates to **true** while the clause x - y < 5 evaluates to **false** is not tested. Thus, potential faults that might occur in this particular situation may not be detected. The next criterion resolves this weakness.

Table 18.11. A test meeting criterion 2

x	y	x - y < 5	x + y > 10
3	4	true	false
1	2	true	false
6	9	true	true
8	1	false	false

Table 18.12. Criterion 3

P_1	P_2	P_3	...	P_n
true	false	false	...	false
false	ture	false	...	false
false	false	true	...	false
...	...	...	...	...
false	false	false	...	true

Criterion 3 Evaluate P_i with T_d to **true** while all other clauses P_1, P_2, ..., P_i -1, P_i + 1, ..., P_n evaluate to **false**, and evaluate P_i to **false** while all P_1, P_2, ..., P_i -1, P_i + 1, ..., P_n evaluate to **true**.

Table 18.12 explains the essential idea of this criterion. Since a true evaluation of each clause P_i (i = 1..n) results in a true evaluation of the entire predicate expression P; even if all the other clauses P_1, P_2, ..., P_ i - 1, P_i + 1, ..., P_n evaluate to **false**, it is definitely useful to test P_i in such a situation. However, for some predicate expressions this criterion may not be applicable. Let us take the predicate expression Q1: x + y > 5 or x + y > 10 as an example to explain this point. When the clause x + y > 10 evaluates to **true** with a test case, it is impossible to evaluate the clause x + y > 5 to **false** with the same test case. Therefore, this criterion cannot be met by any test set. In this case, as mentioned before, this criterion can be used only as a guideline for test case generation, rather than as a strict standard. In fact, all of these criteria can be used flexibly in practice depending on the testing target: they can be used in combination or independently.

Criterion 4 When evaluating a disjunctive clause P_i (i = 1..n) with T_d to **false**, evaluate each Q_j (j = 1..m) to **false** at least once, respectively.

Table 18.13 illustrates the idea of this criterion. In this criterion, when Q_j evaluates to **false**, there is no specific requirement for the evaluations of all the other conjunctions Q_1, Q_2, ..., Q_j - 1, Q_j + 1, ..., Q_m. Another stronger criterion is given next.

Table 18.13. Criterion 4

Q_1	Q_2	Q_3	...	Q_m
true	true	true	...	true
false	*	*	...	*
*	false	*	...	*
*	*	false	...	*
*	*	*	...	false

Table 18.14. Criterion 5

Q_1	Q_2	Q_3	...	Q_m
true	true	true	...	true
false	true	true	...	true
true	false	true	...	true
true	true	false	...	true
true	true	true	...	false

Criterion 5 When evaluating a disjunctive clause P_i (i = 1..n) with T_d to **false**, evaluate each Q_j (j = 1..m) to **false** at least once, respectively, while all the other disjuncts Q_1, Q_2, ..., Q_j - 1, Q_j + 1, ..., Q_m evaluate to **true**.

The idea of this criterion is explained by Table 18.14. It is worth noting that this criterion may not be applicable completely to some predicate expressions. For example, to evaluate the expression x >10 **and** x > 5 to **false**, it is impossible to evaluate x > 10 to **true** when evaluating x > 5 to **false**. In this case, **Criterion** 4 can be applied instead.

18.4 Integration Testing

Since processes are integrated into a CDFD in specifications, the correctness of CDFD depends not only on the correctness of each process, but also the consistency between the interfaces of the processes. Therefore, testing only processes is apparently insufficient in detecting faults existing in CDFDs. Integration testing aims to uncover faults leading to the violation of the consistency between processes in CDFDs. Apart from the verification of the consistency between processes in the same CDFD, it is also important and necessary to ensure the consistency between a process and its decomposition.

A sensible strategy for integration testing of a CDFD is to test every construct contained in the CDFD in order to cover all the possible paths, where a path is a sequence of data flows from a starting node to a terminating node, as defined in Section 5.3 of Chapter 5. As with the unit testing, the approach we take in testing a CDFD is first to derive a proof obligation
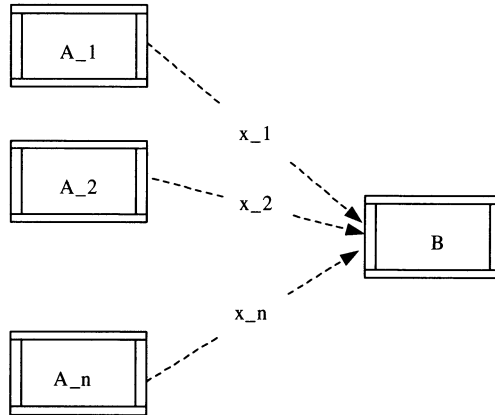
Fig. 18.2. A sequential construct

for ensuring the consistency between processes and then to test the proof obligation.

Basically, there are two kinds of constructs in CDFDs: *sequential* and *conditional constructs*. The focus in testing these constructs is on the derivation of the proof obligations as the testing targets. Verifying the proof obligations by testing can be done by taking an approach similar to that of unit testing.

18.4.1 Testing Sequential Constructs

Let $A\_1, A\_2, ..., A\_n$, and B denote processes. Let $x\_1, x\_2, ..., x\_n$ be output data flows of $A\_1, A\_2, ..., A\_n$, respectively, and the input data flows of B as well. Then $A\_1, A\_2, ..., A\_n$, and B form a sequential construct, as depicted in Figure 18.2.

According to the semantics of a process, when all of $x\_1, x\_2, ..., x\_n$ are available, the precondition of B must evaluate to **true** on these input data flows, because it is an assumption for the postcondition of the process to hold after an execution. A condition for the assurance of such a consistency between $A\_1, ..., A\_n$ and B is formalized as

(pre_A_1 and post_A_1(x_1)) and ... **and**
(pre_A_n and post_A_n(x_n))
=> pre_B

where each **post_A_i(x_i)** $(i = 1..n)$ is a sub-logical expression of the postcondition **post_A_i** of process A_i that contains variable x_i $(i = 1..n)$. For example, let **post_A_1** denote the predicate expression: $x\_1 > a$ **and** $x\_1 <$ 10 **or** $x\_1 > a + 10$ **or** $y < a$, where a is an input constrained by the precondition **pre_A_1**. Then **post_A_1(x_1)** $= x\_1 > a$ **and** $x\_1 < 10$ **or** x_1

> a + 10. Note that **pre_B** does not necessarily involve all x_i syntactically (e.g., **pre_B = true**).

A test oracle is derived from the consistency condition given above:

forall[x_1: T_x_1, x_2: T_x_2, ..., x_n: T_x_n] |
(**pre_A_1 and post_A_1(x_1)) and** ... **and**
(**pre_A_n and post_A_n(x_n)) => pre_B**   (Seq-oracle)

where each T_x_i (i = 1..n) denotes the set of test values generated for variable x_i in the test.

Let us take the sequential construct in Figure 18.2 as an example to see how such a construct can be tested by applying the procedure introduced above. Assume n = 3, and process A_1, A_2, A_3, and B are defined as follows:

process A_1() x_1: int
post x_1 = 5
end_process;

process A_2() x_2: int
post x_2 > 10
end_process;

process A_3() x_3: int
post x_3 = 20
end_process;

process B(x_1, x_2, x_3: **int)**
pre x_1 + x_2 + x_3 < 30
end_process;

A test is given in Table 18.15, where

post_conj = **post_A_1 and post_A_2 and post_A_3**
imp   = post_conj => **pre_B**

Analyzing this test based on the test oracle (Seq-oracle), we can easily conclude that this test is a successful test, because of the first three test cases and their test results. This implies that there is a fault existing in this sequential construct leading to the violation of the consistency condition.

In comparison with the testing of processes, the test oracle of a sequential construct is deterministic in deciding its success or failure. That is, we can definitely determine, by its test oracle, whether a test of a sequential construct is a successful test or failed test.

Table 18.15. A test for the sequential construct

x_1	x_2	x_3	post_conj	pre_B	imp
5	15	20	true	false	false
5	11	20	true	false	false
5	30	20	true	false	false
4	9	20	false	false	true
5	9	12	false	true	true

(a) Single condition structure

(b) Binary condition structure

(c) Multiple condition structure
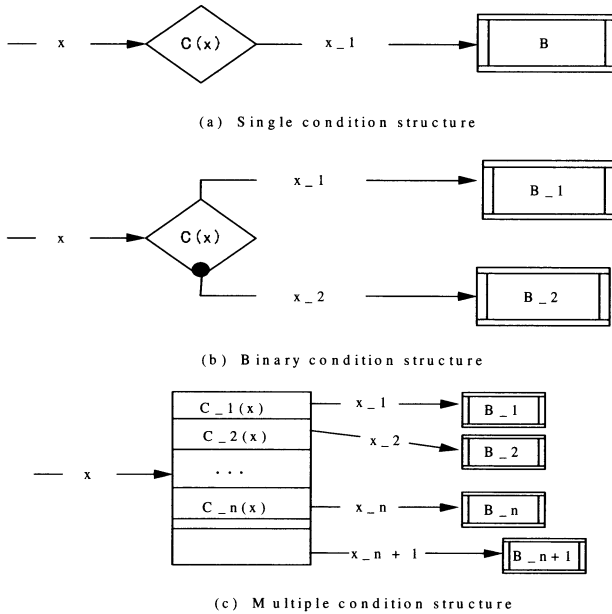
Fig. 18.3. Three conditional constructs

18.4.2 Testing Conditional Constructs

There are three kinds of conditional constructs in SOFL: single condition structure, binary condition structure, and multiple condition structure, as shown in Figure 18.3. Since the testing of these constructs shares the same procedure as for sequential constructs, we do not repeat the discussion on how to generate tests and how to analyze the test results. Instead, we only give the proof obligations for the assurance of the consistency of the constructs, and the derived test oracles for test results analysis.

Single Condition Structures

A construct of this kind is illustrated by Figure 18.3(a). The proof obligation for ensuring the consistency of the construct is

pre **and** post(x) **and** C(x) $=>$ **pre_B**

where pre **and** post(x) denote the pre and postconditions (in fact, only the part containing variable x) of the preceding process of the condition node C(x), respectively, which produces data flow x as its output; **pre_B** is the precondition of process B.

A test oracle for determining whether a test of the single condition structure is successful or has failed is derived from the proof obligation and given as follows:

forall[x: T_x] | pre **and** post(x) **and** C(x) $=>$ **pre_B**

where T_x is the set of values generated for variable x in the test.

Binary Condition Structures

This kind of construct is depicted by Figure 18.3(b). The proof obligation for ensuring the consistency of the construct includes the following two conditions:

pre **and** post(x) **and** C(x) $=>$ **pre_B_1**

pre **and** post(x) **and not** C(x) $=>$ **pre_B_2**

If C(x) is true, the pre and postconditions of the preceding process must, in conjunction with C(x), imply the precondition of process B_1. Otherwise, if C(x) is false, the conjunction of pre, post(x), and negation of C(x), must imply the precondition of process B_2.

A test oracle derived from this proof obligation also includes the two conditions

forall[x: T_x] | pre **and** post(x) **and** C(x) $=>$ **pre_B_1**

forall[x: T_x] | pre **and** post(x) **and not** C(x) $=>$ **pre_B_2**

where T_x, pre, and post(x) have the same interpretations as those given previously.

Multiple Condition Structures

A multiple condition structure is depicted by Figure 18.3(c). The proof obligation includes several conditions:

pre **and** post(x) **and** C_i(x) $=>$ **pre_B_i**   (i = 1..n)

pre **and** post(x) **and**

not (C_1(x) **or** C_2(x) **or** ... **or** C_n(x)) =>
pre_B_n + 1

The test oracle for testing a multiple condition structure of such a kind is derived from this proof obligation:

forall[x: T_x] | pre **and** post(x) **and** C_i(x) => pre_B_i   (i = 1..n)

forall[x: T_x] | pre **and** post(x) **and**
          not (C_1(x) **or** C_2(x) **or** ... **or** C_n(x)) =>
          pre_B_n + 1

18.4.3 Testing Decompositions

As we have discussed in Section 5.4 of Chapter 5, when a process is decomposed into a CDFD, it is desirable to ensure the correctness of the decomposition. That is, for every input of the process, if its precondition holds, then its postcondition must hold on the outputs generated by its decomposition (i.e., the decomposed CDFD). Let P and G denote a process and its decomposition, respectively. Then the correctness of the decomposition is ensured if and only if the following conditions are satisfied:

pre_P => **pre_G**
pre_P and post_G => **post_P**

where **pre_G** and **post_G** denote the pre and postconditions of CDFD G, respectively.

If we restrict the range of values for the input and output variables of process P and other related processes contained in G to the test set used in a test, a test oracle can be derived, as we did for sequential constructs and conditional constructs in previous sections.

We try to avoid in this section a general discussion on how to test the correctness of a CDFD against its high level process, because it may be hard for the reader to understand the important idea. Instead, we explain the method for testing the correctness of a CDFD with a comprehensible example.

Figure 18.4 gives two CDFDs, where process A in CDFD (a) is decomposed into CDFD (b). Assume the processes A, A1, A2, A3, and A4 are defined as follows:

process A(x: **seq of nat0**) y, z: **nat0**
pre len(x) > 0
post y < len(x) **and** z < len(x)
decom G
end_process;

(a)

(b)
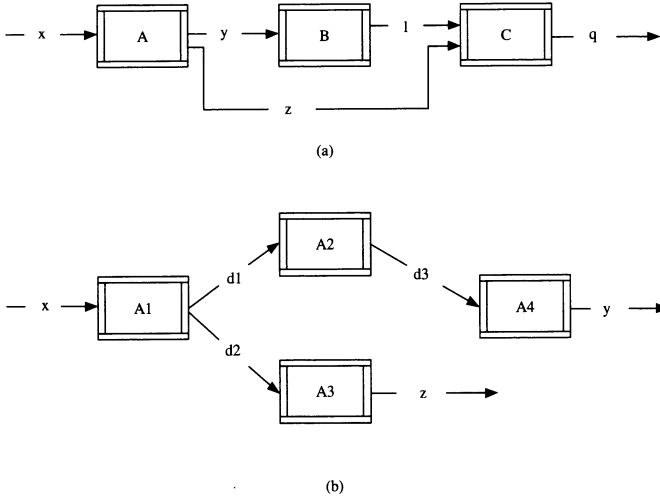
Fig. 18.4. An example of process decomposition

process A1(x: **seq of** nat0) d1, d2: **seq of** nat0
pre   len(x) > 0
post forall[a: **elems**(d1)] a < 60 **and**
      forall[b: **elems**(d2)] | b >= 60 **and**
      union(**elems**(d1), **elems**(d2)) = **elems**(x)
end_process;

process A2(d1: **seq of** nat0) d3: **seq of** nat0
post d1 = [ ] **and** d3 = [ ] **or**
      d1 <> [ ] **and subset**(**elems**(d3), **elems**(d1)) **and**
      forall[e: **elems**(d3)] | e >= 40
end_process;

process A3(d2: **seq of** nat0) z: nat0
post   z = len(d2)
end_process;

process A4(d3: **seq of** nat0) y: nat0
post   y = len(d3)
end_process;

In this particular case, to test the correctness of CDFD (b) with respect to process A, we are required to show that the following conditions hold under the test:

Table 18.16. A test for condition (1)

x	pre_A	pre_A1	condition 1
[35, 90, 85, 39]	true	true	true
[35, 85, 95]	true	true	true
[ ]	false	false	true
[28, 60]	true	true	true

Table 18.17. A test for condition (2)

x	d1	d2	conj1	conj2	condition 2
[35, 90, 85, 39]	[35, 39, 85]	[90]	false	true	true
[35, 85, 95]	[35]	[85, 95]	true	true	true
[ ]	[ ]	[ ]	false	true	true
[28, 60]	[28]	[60]	true	true	true

Table 18.18. A test for condition (3)

d1	d3	pre_A2	post_A2	pre_A4	condition 3
[35, 39, 85]	[85]	true	true	true	true
[35]	[ ]	true	true	true	true
[ ]	[ ]	true	true	true	true
[28]	[ ]	true	true	true	true

(1) **pre_A => pre_A1**
(2) **pre_A1 and post_A1 => pre_A2 and pre_A3**
(3) **pre_A2 and post_A2 => pre_A4**
(4) **(pre_A1 and post_A1) and (pre_A3 and post_A3) and (pre_A4 and post_A4) => post_A**

The testing of these four conditions can be carried out separately, but the consistency of the test cases used in the tests must be guaranteed in order to ensure that these four conditions are accurately tested. For example, the test cases for the input variable x of process A must be the same as those for the input variable x of process A1 in testing condition (1), and the same test cases must also be used in testing condition (2). The reason for this is that, for the same input of process A, we want to find out whether there is any fault in producing its output by its decomposition G.

The tests for testing conditions (1), (2), and (3) are given in Table 18.16, 18.17, and 18.18, respectively. A test for condition (4) is given in Table 18.19 and 18.20. In Table 18.17, conj1 and conj2 are defined as

conj1 == **pre_A1 and post_A1**
conj2 == **pre_A2 and pre_A3**

In Table 18.20, conj3, conj4, and conj5 are defined as follows:

Table 18.19. The first part of the test for condition (4)

x	d1	d2	d3	z	y
[35, 90, 85, 39]	[35, 39, 85]	[90]	[85]	1	1
[35, 85, 95]	[35]	[35]	[ ]	1	0
[ ]	[ ]	[ ]	[ ]	0	0
[28, 60]	[28]	[28]	[ ]	1	0

Table 18.20. The second partt of the test for condition (4)

conj3	conj4	conj5	post_A	condition 4
false	true	true	true	true
true	true	true	true	true
false	true	true	false	true
true	true	true	true	true

conj3 = (**pre_A1 and post_A1**)
conj4 = (**pre_A3 and post_A3**)
conj5 = (**pre_A4 and post_A4**)

Since all the four conditions hold under these tests, no fault is detected. Of course, to make this claim more trustable, more test cases are needed. However, with the increase of test cases, the testing process may become more complicated, and the management of test cases, test targets, and test results may become a serious problem to be resolved. A possible solution to these problems is automation of every activity involved in the testing, usually supported by powerful test tools. Such tools are expected to provide assistance for test case generation, test case optimization (e.g., selecting only the representative test cases from the generated test cases), predicate evaluation, test results analysis, and the management of tests. A prototype tool for testing SOFL specifications has already been developed by our research group, but we need to put more effort for developing it into a useful product.

18.5 Exercises

1. Answer the following questions:
   a) What is a test case ?
   b) What is a test set ?
   c) What is a test suite ?
   d) What is a test target ?
   e) What are possible ways of generating test cases ?
   f) What are the three steps for testing a specification ?
   g) What is a failed test, a successful test, and an uncertain test ?
   h) Is it possible to have a successful test for a process ? If so, give an example. If not, explain why.

2. Generate a test based on **Criterion** 2 given in Section 18.3 for process A1 in Figure 18.4.

3. Try to generate a test based on **Criterion** 3 given in Section 18.3 for the process A2 in Figure 18.4; **Criterion** 2 can be used when **Criterion** 3 is not applicable.

4. Generate a different test from the one given in Section 18.4.3 for the verification of consistency between process A and its decomposition (or the correctness of the decomposition with respect to process A) given in Figure 18.4.

19

Transformation from Designs to Programs

Transformation from design specifications to programs is an activity of constructing programs in a programming language that satisfy the specifications in a specification language; it does not take only the semantics of both specifications and programs into account, as the refinement approach usually emphasizes, but also considers the syntactical change from one language to another.

Transformation is a very important issue to address, because the ultimate goal of software development is to achieve a satisfactory and executable program system. In this chapter, we discuss the techniques for transforming design specifications to programs. As we have learnt in the previous chapters, a design specification may contain several level components, such as modules and classes, CDFDs, and processes and methods. A CDFD represents the behavior of a module, and may contain other kinds of structures for describing complex systems, such as conditional and diverging structures. Each process in a CDFD, as well as each method in a class, is defined with an implicit specification, which is although not desirable at the end of the design phase, or with an explicit specification, or with a mixture of both. Apart from these functional components, data structures defined with various abstract data types, such as set and sequence types, are also involved in the specifications to represent data resources necessary for processes or methods to manipulate. Therefore, transformation from a design specification to a program must take both data and functional components into account.

Usually, a design specification can be transformed into a functionally equivalent program in any kind of high level language, such as Pascal, C, C++, and Java. But, as we mentioned in Chapter 1, object-oriented programming is effective in helping implement good qualities, such as maintainability, in programs. We choose Java as the target language for the transformation of design specifications. That is, the discussions in this chapter are all about how to transform various data and functional components of design specifications in SOFL into Java programs.

Since design specifications are an abstraction of implementations, they usually provide freedom to choose an appropriate implementation strategy.

For this reason, there may be more than one way of transformation. The solution provided in this chapter is only one of them, expected to convey the general guidelines for transformation, from which the reader may derive his or her own way of transformation.

19.1 Transformation of Data Types

Transformation from an abstract data type in specification into a concrete data type in program requires both semantics preservation and syntactic changes. Let T_a and T_c denote an abstract data type and concrete data type, respectively. Semantically, when T_a is transformed into T_c, all the elements of T_a must be represented by the elements of T_c, that is, T_c should contain sufficient elements to represent all the elements of T_a, for this will eliminate the possibility of inappropriate data structures causing the program using T_c not to satisfy the required functions defined in the specification.

Formally, this means that there must exist a retrieve function from T_c to T_a, as explained in Section 14.6 of Chapter 14, which should satisfy the condition

forall[a: T_a] **exists**[c: T_c] | Retr(c) = a   (data-tran)

Transformation of a data type does not need only to conform to this rule, but need also to take other related issues into account, such as what the data type is used for and how easily the related built-in operators can be implemented, and so on. For example, a sequence of integers, that is, **seq of int**, can be transformed into an *array* of integers, a *vector*, or a *sequential file* in Java; it can also be directly transformed to a *list*, depending on how it is used in the program. In general, the choice of the concrete data types in the transformation will affect somehow the algorithms of the implemented program using the data types. Therefore, it is essential to strike a balance between data structures and algorithms.

Under the constraint of the rule (data-tran), A suggested transformations of all the built-in data types in SOFL are given in Table 19.1.

Java is a powerful language because of its rich data structures, supported by related classes available in the class libraries. The transformation of most abstract data types in the table is quite straightforward, because they can be implemented by similar concrete data types available in Java. The only transformations that may puzzle the reader are those of composite and product types: both are transformed to proper classes. This is in fact not that difficult to understand, as long as one makes a comparison between a composite type or a product type and a class with no methods. For example, consider the composite type C:

Table 19.1. Transformation of data types

data type in SOFL	data type in Java
nat0	int
nat	int
int	int
real	double
bool	boolean
char	char
Enumeration	Enumeration or array of String
set type	Set, array, vector, or file
seq type	List, array, vector, or file
string	string
map type	Map, array, vector, or file
composite type	class
product type	class
union type	Object class

```
C = composed of
    f_1: T_1
    f_2: T_2
    f_3: T_3
    end
```

By transforming each field of the composite type C to an instance variable (or attribute variable) of the corresponding class in Java, we get

```
class C {
    T_1 f_1;
    T_2 f_2;
    T_3 f_3;
}
```

where the types T_1, T_2, and T_3 are assumed to have been transformed either to a class or to basic types of Java. Note that the syntax of variable declaration in Java differs from that of variable declaration in SOFL. The former gives the type prior to the variable, with a space in between, while the latter lets the variable appear before the type, with a colon separating them. Of course, methods may be defined in the class C, if necessary.

19.2 Transformation of Modules and Classes

A module or class in specifications, known as *source module* or *class*, has a structure similar to that of a class in Java, known as *target class*. Therefore,

transformation

module m	**class** m {
store *s*	**instance variable** *s*
data flows z,u,v,t,w	**attributes** z,u,v,t,w
process *Init*	**constructor** m
CDFD	**method** *G*
process *A*	**method** *A*
process *B*	**method** *B*
process *C*	**method** *C*
function F	**method** F
	}
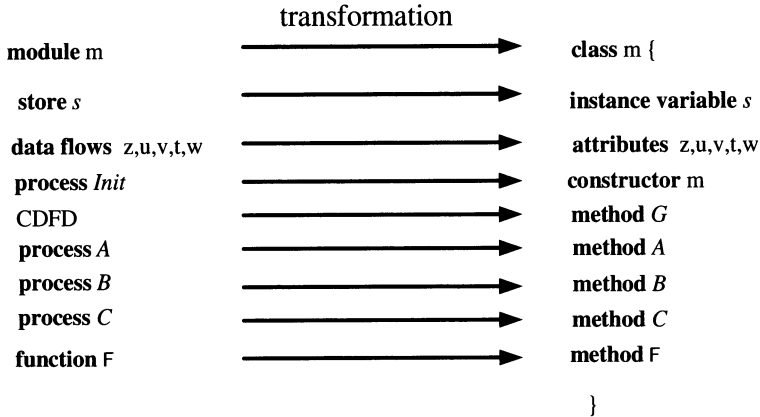
Fig. 19.1. An illustration of the main ideas of the tranformaqtion guidelines

it is quite natural and convenient to transform a source module or class to a target class. Although their transformations share the similarity due to their similar structures, source classes are much more straightforward than modules to transform to target classes. The reason for this is that source classes share the same concepts as target classes, whereas source modules have additional features, such as CDFDs and exclusive inputs and/or outputs, which need special treatment during transformations.

The underlying guidelines for the transformation of a source module to a target class are summarized as follows:

- Transform the module name to the class name.
- Transform a constant declaration to a constant declaration (using the keyword final prior to the constant variable in the declaration).
- Transform a type declaration to either a corresponding basic type or a proper target class, as described in Table 19.1.
- Transform a store variable declared in the **var** part to an instance variable (or field) of the class.
- Transform a source process (a process of the source module) to a target method (a method of the target class).
- Transform a function, if any, defined in the module to a target method that does not change the state of the target class.
- When transforming a process or a function, make sure that the invariants defined in the **inv** part of the module are not violated.
- Transform the CDFD of the module to a target method of the class in which all the related methods are integrated through their invocations.

Figure 19.1 illustrates the main ideas of these guidelines.
These guidelines are also applicable to transformations of source classes, except that superclasses need to be taken into account in the transformations.
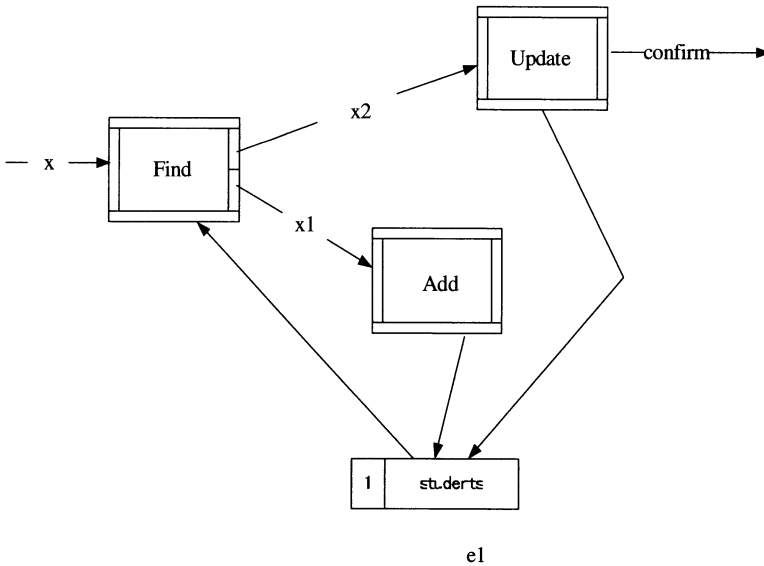
Fig. 19.2. The CDFD of module Student_Management

That is, the class hierarchy in the specification must be transformed to a proper class hierarchy in the program. Furthermore, there is no need to take into account the CDFD of a source class, if it is given, in the transformation, because semantically it does not represent anything additional, but is just "syntactical sugar" to depict the source methods and their relations with the state variables of the class.

Consider the transformation of the module **Student_Management** and the class **Student**, given below, as an example. We assume that the module **Student_Management** is a decomposition of a process defined in the high level module **Faculty_System**, which is presumably defined somewhere in the specification. The CDFD given in Figure 19.2 describes the behavior of the module **Student_Management**: when a student x is available, the process **Find** tries to find x in the store **students**; If x exists, it is passed to process **Update** through x2; otherwise, x is passed to the process **Add** through variable x1.

Furthermore, a class **Student** is defined as a subclass of an already defined class **Person**, and used for defining the type **Students** as a set of **Student** objects.

```
module Student_Management / Faculty_System;
const
 PI = 3.14159;
type
 Students = set of Student;
var
```
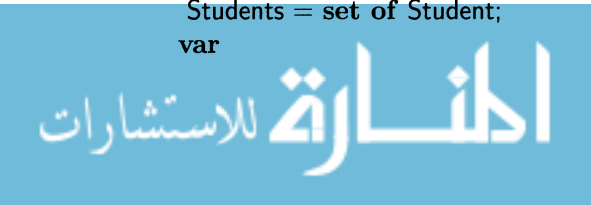
```
 students: Students;
inv
 forall[x: Student] | x.total_credit <= 135;
behav CDFD_e1;

process Init()
post students = { }
end_process;

process Find(x: Student) x2: Student | x1: Student
ext rd students
post if (exists[y: students] | x.id = y.id)
      then x2 = x
      else x1 = x
end_process;

process Add(x1: Student)
ext wr students
post students = union(~students, {x1})
end_process;

process Update(x2: Student) confirm: bool
ext wr students
post if x2 inset students
      then students = diff(~students, {x2}) and confirm = true
      else students = union(~students, {x2}) and confirm = false
end_process;
end_module;


class Student / Person;
var
 id: string;
 name: string;
 total_credit: nat0;

method Init()
post id = 0 and name = "" and total_credit = 0
end_method;

method Set_Name(name1: string)
ext wr name
post name = name1
end_method;
end_class;
```

Applying the guidelines introduced previously, we transform the source module Student_Management into the following target class:

```
public class StudentManagement {
static final double PI = 3.14159;
private Set students;

public Student_Management() {
 students = new HashSet();
}

public boolean Find(Student x) {
if (students.contains(x))
return true;
else
return false;
}

public void Add(Student x) {
students.add(x);
}

public boolean Update(Student x2) {
//create an array objs containing all the elements of the set students.
Student objs[] = students.toArray();
int control = 0;
for (int i = 0; i < objs.length; i++)
  if (objs[i].getID() == x2.getID())
 {
    //Remove the identified element from the set students
    students.remove(objs[i]);
   control = 1;
    break;
 }
 //Add the element x2 to the set studnets
if (control == 1) {
 students.add(x2);
 return false;
 }
else return true;
 }

//A method derived from the CDFD of the module in the specification
 public boolean CDFD(Student x) {
if (Find(x))
```

```
 Add(x);
else
 return Update(x);
}
}
```

The class name **StudentManagement**, conforming to the convention of Java identifiers, is derived from the source module name **Student_Management**. The constant **PI** is declared in the same way as it is declared in the specification. It is quite interesting to see that the type **Students** defined as a set of **Student** in the source module is no longer necessary in the target class; instead, the store variable **students** in the source module is directly transformed to the instance variable **students** declared with the class **Set** in the target class **StudentManagement**. Since the invariant in the module only imposes a constraint on the type **Students**, and therefore on variable **students**, it does not correspond to any construct in Java. However, it must be ensured that the invariant is not violated by the related methods. The constructor **Student-Management()**, which is required to share the same name as the target class, is defined based on the process **Init** in the source module. The methods **Find**, **Add**, and **Update** are all derived from the corresponding processes in the source module, but the parameters and types of these target methods may have been modified to properly fit into the context of the program. The important thing in the transformation is not to forget to transform the CDFD of the source module to a method in the target class. In our transformation, the CDFD of the module is transformed to the method CDFD (of course, the method can be named differently) of the target class. The body of this method is an implementation of the CDFD of the source module.

The target class **Student** used in the class **StudentManagement** needs to be defined by transforming the source class **Student**. Since all the instance variables of **Student** are required to be private, methods necessary for accessing those instance variables, such as **getID**, must be defined in the target class. Thus, the source class **Student** is transformed into the following target class:

```
public class Student extends Person {
 int id;
 string name;
 int totalCredit;

public Student() {
  id = 0;
  name = "";
  totalCredit = 0;
}
 public void SetName(String name1) {
  name = name1;
}
```

```
    public int getID() {
     return id;
    }
    public String getName() {
     return name;
    }
    public int getTotalCredit() {
     return totalCredit;
    }
    }
```

The transformation from the source class **Student** to the target class with the same name is quite straightforward: mapping from attribute variables to instance variables, from the constructor **Init** to the constructor **Student()**, and from source methods to target methods. To supply necessary functionality, additional methods may be defined in the target class. The methods **getID**, **getName**, and **getTotalCredit** in the class **Student** are the methods of this kind, through which the attributes of the objects of the class can be accessed.

19.3 Transformation of Processes

We have seen the transformation of some processes in the module **Student_Management** discussed in the preceding section, which dealt only with some examples. In fact, a general transformation strategy for processes can be difficult, because the correctness, efficiency, structure, and conciseness of the program must be taken into account in a transformation. Since most of these issues are still under research, and there is no clear solution yet, we will discuss the transformation only from the correctness point of view. That is, the guidelines to be given are all for the assurance of the semantic correctness of the program against its specification, without consideration of other factors.

As mentioned in the underlying guidelines before, a process in a source module is usually transformed to a method in the target class. However, there is a difference between a process in a module and a method in a Java class: a process allows exclusive inputs and outputs, whereas a method does not. How to resolve this difference in transformation is an important issue to address. In this section, we first discuss the guidelines for transforming a process with no exclusive inputs and outputs, and then proceed to explain how to tackle the exclusive inputs and outputs problem.

19.3.1 Transformation of Single-Port Processes

A single-port process is a process with only one input port and one output port. For example, the processes **Add** and **Update** in Figure 19.2 are both

single-port processes, whereas the process Find is not a single-port process because it has two output ports. In general, a single-port process, say A, has the form

> **process** A(x_1: Ti_1, x_2: Ti_2, ..., x_n: Ti_n)
>   y_1: To_1, y_2: To_2, ..., y_m: To_m
> **pre pre_A**
> **post post_A**
> **end_process**

where Ti_k (k = 1..n) are types for input variables and To_j (j = 1..m) are types for output variables.

There are two ways to transform this process to a target method. One is to take the precondition into account, which usually results in a robust implementation, and the other is to completely ignore the precondition, which will possibly lead to a non-robust (but correct) implementation.

Taking the precondition into account, we transform process A into method A in the target class Transformation1

```
class Transformation1 {
 To_1 y_1;
 To_2 y_2;
 ...
 To_m y_m;
 ...
 public void A(Ti_1 x_1, Ti_2 x_2, ..., Ti_n x_n) {
  if (pre_A)
  {
    Tran(post_A)
  }
  ...
 }
```

where Tran(**post_A**) denotes the program segment generated from **post_A**.

The variables y_1, y_2, ..., y_m that are originally declared as the output variables of the process A are declared as the instance variables of the target class, because a method in a Java class does not allow more than one output. The reason that this transformation is robust is that the precondition of process A is always checked before the execution of the program segment resulting from the transformation of **post_A**. If the precondition is false, then no program code (or, alternatively, the code segment for generating error messages) is executed, which satisfies the requirement of the process specification. If **pre_A** is a more complex predicate expression, it may first need to be transformed into a proper program code, evaluating it to a truth value, and then

the result is used in the conditional statement. If process A is given in an explicit form, such as

```
process A(x_1: Ti_1, x_2: Ti_2, ..., x_n: Ti_n)
          y_1: To_1, y_2: To_2, ..., y_m: To_m
pre pre_A
explicit
 S
end_process
```

then, by taking the same strategy, we transform it into method A of the target class Transformation2:

```
class Transformation2 {
 To_1 y_1;
 To_2 y_2;
 ...
 To_m y_m;
 ...
 public void A(Ti_1 x_1, Ti_2 x_2, ..., Ti_n x_n) {
  if (pre_A)
  {
    Tran(S)
  }
  ...
 }
```

The only difference between this transformation and the previous one is the use of Tran(S), the program segment produced from the explicit specification S, to replace Tran(post_A).

Another transformation strategy is to take only the postcondition of the process into account. The reason we can ignore the precondition of the process is that the precondition is only an assumption for the postcondition to hold after the execution of the process. That is, the precondition is assumed to be ensured by the environment before available input data flows are sent to the process, and the process is responsible for providing outputs satisfying the postconditions only for those inputs satisfying the precondition.

Let us consider the implicit specification of process A again as an example. It is transformed into the method A in the target class Transformation3:

```
class Transformation3 {
 To_1 y_1;
 To_2 y_2;
 ...
 To_m y_m;
```

```
   ...
  A(Ti_1 x_1, Ti_2 x_2, ..., Ti_n x_n) {
     Tran(post_A)
   }
   ...
  }
```

Although this transformation generates a correct method A in the sense that it satisfies its process specification, the method is less robust than the one resulting from transforming the process A taking its precondition into account. The reason for this is that method A is not capable of dealing with the exceptional cases when the inputs do not satisfy the precondition; it may cause abnormal termination of the program or undesirable results. This is especially possible when the method directly takes inputs from a graphical user interface, where it is usually difficult to ensure that the inputs satisfy the required precondition. Therefore, the transformation involving the precondition given before is usually a better choice, although it may sacrifice a little time efficiency due to the necessity of evaluating the precondition in the program.

19.3.2 Transformation of Multiple-Port Processes

A process with multiple input or output ports allows exclusive input or output data flows. In order to focus on the issue of how to deal with the multiple-ports problem in process transformation, rather than on the issue of dealing with the number of ports, we take the process B with two input and output ports, respectively, as an example to explain the transformation strategy.

Let B be defined as

```
process B(x_1: Ti_1 | x_2: Ti_2) y_1: To_1 | y_2: To_2
pre pre_B
post post_B
end_process
```

Assume that output y_1 is generated based on input x_1, and y_2 is based on input x_2. A simple way to transform this process is to define two methods in the target class, for example, B1 and B2. The method B1 takes x_1 as input and generates y_1 as output, while B2 takes x_2 and produces y_2. Thus, we get the target class AnotherTransformation:

```
class AnotherTransformation {
  ...
  public To_1 B1(Ti_1 x_1) {
     if (pre_B(x_1))
     Tran(post_B(y_1))
  }
```

```
public To_2 B2(Ti_2 x_2) {
 if (pre_B(x_2))
 Tran(post_B(y_2))
}
...
}
```

In this transformation, **pre_B(x_1)** denotes the subexpression of the precondition of process B that is intended to constrain x_1 but not x_2, while **pre_B(x_2)** is another subexpression of the precondition constraining x_2 but not x_1. For example, suppose **pre_B** = x_1 > 0 **and** x_2 > 0, then **pre_B(x_1)** = x_1 > 0 and **pre_B(x_2)** = x_2 > 0. Likewise, **post_B(y_1)** and **post_B(y_2)** can be interpreted similarly, but within the context of the postcondition of process B. Thus, when x_1 is supplied, method B_1 will be invoked to provide the required functionality, whereas when x_2 is supplied, B_2 will be invoked.

Another possible transformation strategy is to implement process B as a class, say ProcessClass, in which the two methods B1 and B2 are defined. Then, in class AnotherTransfromation, an object of class ProcessClass, say processOBJ, is instantiated to allow possible invocations of the methods B1 and B2 through processOBJ in the method implementing the corresponding CDFD in class AnotherTransformation. As the result, we may produce the following classes:

```
class AnotherTransformation {
 ...
public void CDFD(...) {
ProcessClass processOBJ = new ProcessClass();
...
processOBJ.B1(x_1);  //invoke method B1 of the object processOBJ
...
processOBJ.B2(x2);  //invoke method B2 of the object processOBJ
...
}
...
}

class ProcessClass {
...
public To_1 B1(Ti_1 x_1) {
   if (pre_B(x_1))
   Tran(post_B(y_1))
 }
 public To_2 B2(Ti_2 x_2) {
 if (pre_B(x_2))
 Tran(post_B(y_2))
```

```
    }
    ...
    }
```

It is important to keep in mind that a process in a CDFD is usually trans-formed to a method invocation (or method call) in the corresponding method implementing the CDFD of the target class. A simple example of transformation will help the reader understand the principle of this transformation strategy.

19.4 Transformation of CDFD

We have mentioned that the CDFD of a module needs to be transformed into a target method of the target class in which all the related methods are integrated according to their invocations. However, we have not systematically discussed anything about the transformation strategy for CDFDs. In this section, we address this problem.

To help the reader focus on the main idea of transformation rather than struggle to understand complicated formal expressions dealing with general cases, we take the approach of using examples, as we did before, to explain the transformation strategy.

A CDFD may contain the the following structures: *sequential, conditional, nondeterministic, broadcasting, parallel* structures. There are three kinds of conditional structures: *single condition structure, binary condition structure,* and *multiple condition structure.* We discuss the transformation of these structures by giving the corresponding guidelines. Before proceeding to introduce the guidelines, we need a function from CDFD structures to algorithms to help in the description of the guidelines.

Definition 45. *Let S_{cdfd} denote the set of all possible CDFD structures and A_{java} the set of all possible algorithms in Java. Then, we let T_c denote the function mapping from S_{cdfd} to A_{java}, that is,*

$$T_c : S_{cdfd} \rightarrow A_{java}$$

Guideline 1 Let S denote the sequential structure in Figure 19.3. Then,
$T_c(\text{S}) ==$ TY_1 y1;
          TY_2 y;
          TY_3 s;
          y1 = A1(x, s); //take x and s to produce
                    //y1 and update s, which is implemented
                    //in the definition of A1
          y = A2(y1, s); //take y1 and s to produce
                    //y and update s, which is implemented
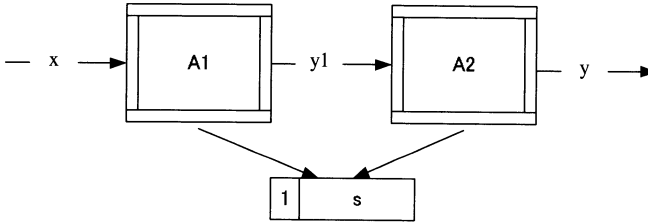   //in the definition of A2

Fig. 19.3. A sequential structure

where we use == to mean "is defined as" and = to mean assignment in Java.

The fundamental idea of the guideline is to treat each process in the CDFD as a method invocation in the corresponding Java program. This underlying idea is shared by the other guidelines to be introduced below. For example, process A1 is invoked first with the actual parameters x and s, and then its result serves as the actual parameter of process A2 when it is invoked. An interesting point is the way of handling store s in the transformation. Since s is accessed by two processes in turn, it is treated as a global variable in the method invocations. The reference variable s provides the initial values for methods A1 and A2, and holds the final values after the invocations of the methods.

Guideline 2 Let S denote the conditional structure in Figure 19.4. Then,
$T_c(\mathsf{S})$ == TY_1 s; //store s is treated as a global variable
        ...   //other related variable declarations
   if (P(y))
    { y1 = y;
     w1 = B(y1);
    }
   else
    { y2 = y;
     w2 = C(y2);
    }

The resulting program reflects the semantics of the conditional structure. If condition P(y) evaluates to true, data flow y is passed to y1, process B is executed, and output data flow w1 is produced based on y1 and s. However, if P(y) evaluates to false, data flow y is passed to y2, process C is executed, and data flow w2 is generated based on y2 and s, and s is possibly updated as well.
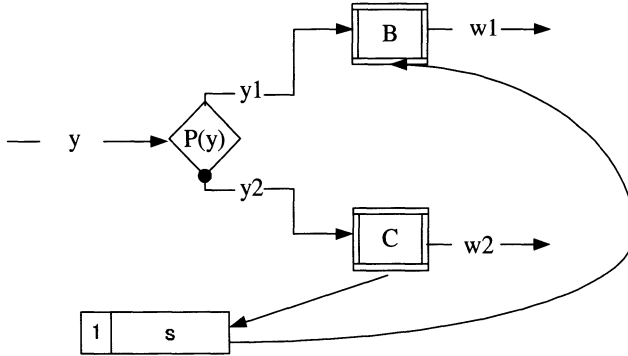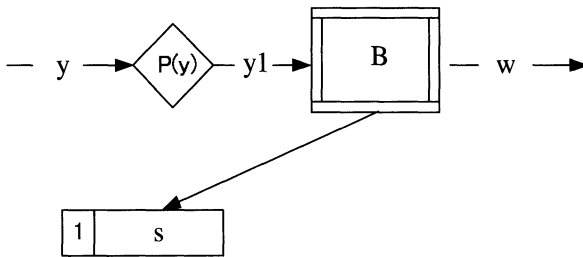
Fig. 19.4. A binary condition structure

Fig. 19.5. A single condition structure

Guideline 3 Let S denote the single condition structure in Figure 19.5. Then,

$T_c(S) ==$ TY_1 s; //store s is declared as a global variable

```
...
if (P(y))
   { y1 = y;
     w = B(y1, s);
   }
```

The generated program implements the semantics of the single condition structure. If condition P(y) evaluates to true, y is passed to y1, and process B is executed to produce w based on y1 and s, and to possibly change the value of global variable s. However, if P(y) is false, y1 is just consumed, without executing any process (that is, the execution of the current structure terminates). In the program, variable w can be treated either as a global variable or as a reference variable of the corresponding method of the structure, for it may hold the final result of the execution.
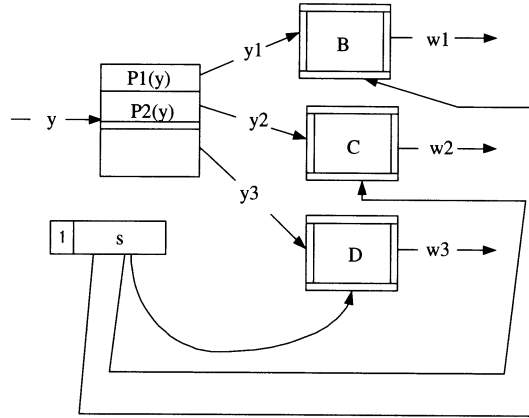
Fig. 19.6. A Multiple condition structure

Guideline 4 Let S denote the CDFD containing a multiple condition structure given in Figure 19.6. Then,

$T_c(S) ==$ TY_1 s; //declaring s as a global variable
...
if (P1(y))
{ y1 = y;
    w1 = B(y1, s);
}
else if (P2(y))
{ y2 = y;
    w2 = C(y2, s);
}
else if (!(P1(y)||P2(y)))
{ y3 = y;
    w3 = D(y3, s);
}

When one of the conditions P1(y) and P2(y) is true, the corresponding process B or C is executed. If both conditions are false, the default process D is executed. In fact, the best program construct for implementing the multiple condition structure may be a *case* statement, but since a similar statement known as *switch* in Java allows decision expressions to be only integral expressions, it is not sufficient to deal with possibly complicated decisions in the multiple condition structure. Therefore, in general *if-then-else* statement can be used to implement the multiple condition structure.
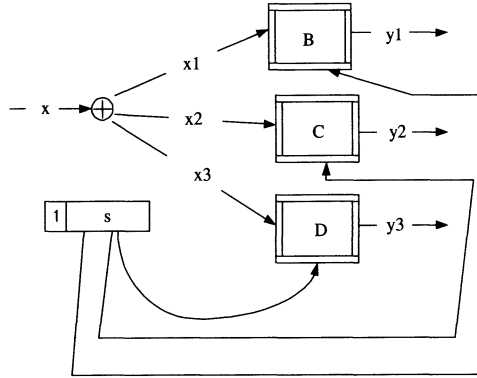
Fig. 19.7. A nondeterministic structure

Guideline 5 Let S denote the CDFD of Figure 19.7 that involves a non-deterministic structure. Then,

$T_c(\mathsf{S}) ==$ TY_1 s; //treating store s as a global variable

```
...
if (P(x))
{ x1 = x;
    y1 = B(x1, s);
}
else if (P1(x))
{ x2 = x;
    y2 = C(x2, s);
}
else if (P2(x))
{ x3 = x;
    y3 = D(x3, s);
}
```

When x is available, it will be transmitted through x1, x2, or x3 to only one of processes B, C, and D in a nondeterministic manner. In the nondeterministic structure, the specific conditions for determining which of processes B, C, and D needs to be executed is not given explicitly; therefore, they need to be given in the program resulting from transformation of the CDFD, such as P(x), P1(x), and P2(x). The conditions must be given in a way that ensures the smooth execution of the CDFD (that is, no deadlock of executions should be created in the program due to the transformation of the nondeterministic structure).
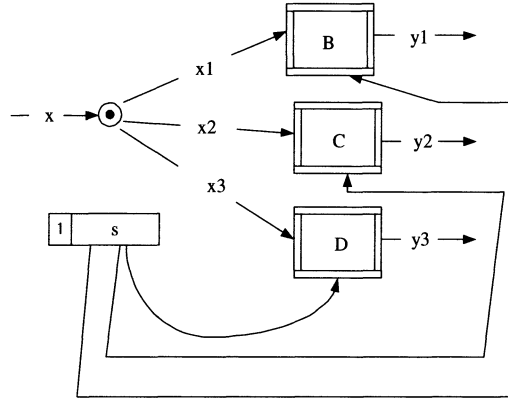
Fig. 19.8. A broadcasting structure

Guideline 6 Let S denote the broadcasting structure in Figure 19.8. Then,

$T_c(\mathsf{S}) ==$ TY_1 s; //s is declared as a global variable

$\quad$ ...
$\quad$ x1 = x;
$\quad$ x2 = x;
$\quad$ x3 = x;
$\quad$ y1 = B(x1, s);
$\quad$ y2 = C(x2, s);
$\quad$ y3 = D(x3, s);

The broadcasting structure is opposite to the nondeterministic structure: when x is available, it will be transmitted to all the processes B, C, and D. Since there is no direct connection between any two of the three processes B, C, and D, and all of them only read data from store s, executions of the three processes do not depend on each other. Therefore, the broadcasting structure can be implemented by a sequence of method invocations in any order. In the guideline, one choice of such a sequence is provided, that is, first invoke method B, then C, and finally D.

Guideline 7 Let S denote the iteration structure in Figure 19.9. Then,

$T_c(\mathsf{S}) ==$ TY_1 s; //declare s as a global variable

$\quad$ ...
$\quad$ y = E(x);
$\quad$ while (P(y)) {
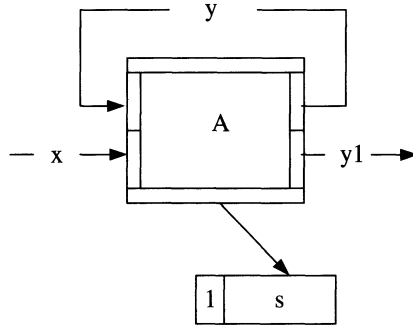$\quad\quad$ y = E1(y);
$\quad$ }
$\quad$ y1 = E2(x, y);

Fig. 19.9. An iteration structure

The transformation of the iteration structure tries to implement its semantics: when x is available, either y or y1 is generated, depending on the related guard condition given in the postcondition of process A; if y is generated, the loop starts until the guard condition becomes false and y1 is produced. This transformation is less straightforward than other transformations given in the previous rules, because the production of data flow y or y1 is actually dependant on the pre and postconditions of process A. Therefore, the resulting program given in the guideline shows only an outline of the transformation, in the sense that the specific expressions E, E1, E2, and condition P(y) are not given precisely; they must be formed by taking the specific pre and postconditions of process A into account.

**Guideline** 8 Let S denote the parallel structure in Figure 19.10, where the executions of process A and B are independent of each other. Then,
$$T_c(S) == \mathsf{TY\_1\ s};\ //\text{store } s \text{ is treated as a global variable}$$
$$...$$
$$y1 = A(x1, s);$$
$$y2 = B(x2, s);$$
Since the executions of processes A and B are independent of each other, although they read data from the same data store s, it is correct to transform the parallel structure into a sequential structure in the program. Of course, the order of invoking methods A and B (corresponding to processes A and B in the CDFD) can be altered due to their independence in execution.

By now we have discussed the guidelines for transforming all the fundamental CDFD structures. The important thing is that we should not treat the introduced guidelines as precise rules for the transformation of CDFDs, because they may not cover all the possible cases in each structure category. In fact, since there are numerous ways to combine the fundamental structures in CDFDs, providing rules to cover general cases can be extremely difficult, if not impossible. Readers who are interested in the transformation of CDFDs are
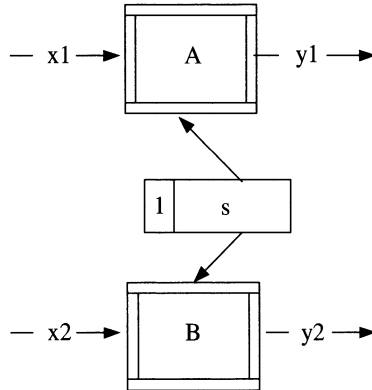
Fig. 19.10. A parallel structure

encouraged to study this issue further to see whether there is any possibility of coming up with general rules for transformation of CDFDs.

19.5 Exercises

1. Give another way of transforming a source module and class that differs from that of the one given in Section 19.2.
2. Give a transformation of process A that is different from the one given in Section 19.3.1 in the sense that the target method A produces an error message when the precondition is not satisfied by the inputs.
3. Give another different transformation of process B, whose format is given in Section 19.3.2, with two input and output ports.
4. Suppose process A is decomposed into a CDFD. Give a transformation of A that utilizes the CDFD in defining the body of the target method.
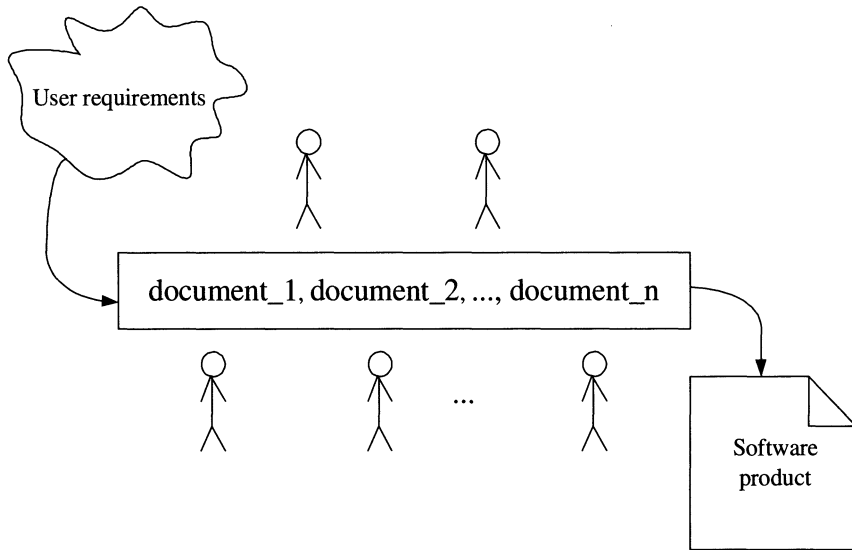
Intelligent Software Engineering Environment

To help people enjoy the benefit of using the SOFL formal engineering method, software support tools are extremely important. The combination of condition data flow diagrams and the textual language for defining their components in the associated modules provides good comprehensibility of the entire specification and allows people at different levels of software projects to work together smoothly, but drawing condition data flow diagrams can be time consuming, especially when the diagrams need frequent changes during the construction of specifications. Furthermore, all the activities involved in the SOFL process, including the capturing of informal user requirements, transformation from informal to semi-formal and then to formal specifications, verification and validation of various level specifications and programs, and process management usually take time and effort and may result in high cost. To resolve these problems, a quality *software engineering environment* for SOFL is necessary and useful. In this chapter, we discuss the issues concerned with the building of software engineering environments.

20.1 Software Engineering Environment

Software Engineering Environment refers to integrated software toolkits in which different tools work together to fulfill software engineering tasks [83]. They are an effective way to enhance the productivity and reliability of software development due to the high speed and large memory capacity of modern computers. Ideally, such an environment should contain software tools supporting every activity in every phase of software engineering, such as requirements analysis, design, transformation, verification, and maintenance, but due to the difficulty and complexity of building such a powerful environment, most of the existing environments concentrate on the support for specific activities. For example, the *IFAD VDM-SL Toolbox* supports the construction and testing of formal specifications using VDM-SL (Vienna Development Method - Specification Language) [110]; *Rational Rose* supports system analysis and

Fig. 20.1. An illustration of production line for software development

design using UML (Unified Modelling Language) [47]; and *JBuilder* supports
the building of program systems using Java [46].

Ideally, a software engineering environment should have the capability of
enforcing a production line for software development, as illustrated in Figure
20.1, to achieve the harmony of software tools, development methods, and
human developers, since such harmony will eventually contribute to high pro-
ductivity and reliability of the final software product. However, such a rigorous
production line still seems impossible under the support of existing software
engineering environments for many reasons, including

- The languages for specifications and designs are mainly informal, which
  imposes a difficulty for high automation in construction, transformation,
  and verification of specifications.
- The environments are implemented in a way that human developers need
  to make decisions on selecting software tools and activities in software
  process. This may create the risk of developers avoiding some necessary
  activities (e.g., specification review) and/or not meeting the required stan-
  dards (e.g., not updating the high level specifications after the low level
  specifications are changed).
- The languages and methods for writing documents at different levels may
  not be coherent; therefore, a powerful support environment is difficult to
  build.

However, since SOFL has properly combined natural language, graphical
notation, and formal textual notation to form a single coherent language with

precise syntax and semantics, a more intelligent software engineering environment for SOFL can be built to overcome the weakness of the present software environments.

20.2 Intelligent Software Engineering Environment

Intelligent Software Engineering Environment, ISEE for short, is a further development of the present software engineering environments toward providing more intelligence in supporting software engineering activities [65]. By intelligence we usually mean the power of perceiving, learning, understanding, and knowing things of interest. Traditionally, intelligence is regarded as the quality which only human beings can posses. The typical way to make a computer intelligent is to write programs that have the knowledge of solving problems of interest, that can conduct reasoning based on the existing knowledge, and that can learn new knowledge from experience. However, since software development itself is an intelligent process involving intelligence in documentation, understanding, communication, creation of system architecture and algorithms, and so on, it seems difficult to realize all the qualities of intelligence in supporting software development. In this section, we intend to discuss potential features of intelligent software engineering environments in general, and possible ways to build them, especially the ones for SOFL. Although the theory and technology for building intelligent software engineering environments are far from mature, we hope the discussions in this section will inspire more researchers, students, and tool builders to develop their interests and technologies in this area.

In fact, a realistic intelligent software engineering environment does not need to be very ambitious. It should focus on the issue of how to help human (developers) provide their best intellectual inputs (e.g., formal specification of a process) necessary for system development, which for the computer is extremely difficult or impossible to provide. The intelligent environment should treat a human being as a special "software tool," and automate all the development and management operations necessary for software projects. The role of a human "software tool" differs from that of other software tools in the sense that human beings provide intellectual inputs that are impossible to obtain in any other way, whereas other software tools provide functionality based on the stored data and algorithms implemented in advance. The distinctive feature of an intelligent environment is that human beings must be *controlled* and *guided* by the environment in developing their systems. This point is quite different from the traditional software engineering environments in the sense that a traditional environment provides only a collection of related software tools that can be freely used by human beings to fulfill their software development tasks. Furthermore, to facilitate the interaction between the computer and human developers, an intelligent environment must provide

a user-friendly interface, allowing developers to input their information by speaking, drawing on the screen, and typing on the keyboard, and so on.

In summary, an intelligent software engineering environment should provide the following functions:

- Human developers are treated as "software tools," and managed by the environment.
- Human developers are guided by the environment to fulfill their tasks, and to proceed from one phase to another.
- Faults in documents are prevented *during* the construction of documents, rather than detected after the construction of documents.
- All the project management operations are automated by the environment.
- A user-friendly interface is provided to help human developers input their information by speaking, drawing, and typing.

To build an intelligent environment, we need the environment to embody necessary knowledge in a knowledge base, which indicates what is to be done under what conditions. Also, the knowledge base should be expanded automatically by learning from human inputs and past experience in building similar software systems. The important role of the knowledge base is to enable the software environment to present *guidance* to human developers, to instruct them what to do next, and to automate management operations, such as linking different parts of a specification, or the corresponding parts in both specifications and programs.

From the internal structure point of view, an intelligent software engineering environment should provide the following mechanisms:

- A *knowledge base* storing all the necessary knowledge for software systems development.
- A *control program* that interacts with both the human developer and the knowledge base, and carries out effective search and application of the relevant rules in the knowledge base to provide accurate and efficient guidance to human developers.
- A *learning program* that automatically collects and builds knowledge from the documentation of previous software projects and the human inputs during the development process.

The knowledge for software development can basically be divided into two categories: *domain-based knowledge* and *method-based knowledge*. The domain-based knowledge is domain specific, encompassing all the necessary knowledge about a specific domain, for example, a banking system, the common architecture of the software systems solving the problems in the domain, the techniques to build specific software systems in the domain based on the common architecture and the knowledge of previous systems, and so on.

The method-based knowledge is method specific, containing all the necessary knowledge about the documentation techniques, the process of proceeding from one stage to another, the software tools and their applications

available in the environment of a specific development method, such as SOFL, and other related operations required by the specific method.

20.3 Ways to Build an ISEE

There are three ways to build an intelligent software engineering environment, based on the two kinds of knowledge about software development discussed in the preceding section, *domain-driven*, *method-driven*, and *the combination of both*.

20.3.1 Domain-Driven Approach

The domain-driven approach to building an ISEE makes use of domain-based knowledge to support the construction of software systems in a specific domain. The key issue in this approach is how to build the knowledge base that contains sufficient knowledge about the domain and provides efficient knowledge retrieving capability.

Since software systems in the same domain usually share common features while each specific system differs from the others, the common features should be captured and expressed as knowledge in a knowledge base. When a specific system in the domain is built, the common features must be properly adopted and the system-specific features must be obtained by both tailoring the common features and using the developer's inputs.

Apart from expressing the common features, the domain-based knowledge base should also contain knowledge about the rules for checking the consistency and completeness of systems, transformation from specifications to designs and programs, and verification and validation of the programs produced.

20.3.2 Method-Driven Approach

The method-driven approach to building an ISEE is based on the construction of a method-based knowledge base. The knowledge base should contain sufficient knowledge about the development method itself, including the syntax and semantics of the language concerned, rules for using the language, and the steps to take to ensure the consistency and validity of documentation at all the possible levels. The most distinctive feature of this approach from the domain-driven approach is that the knowledge base usually contains no knowledge about specific domains, but only the one about the specific method.

Two levels of knowledge can be supported by this kind of ISEE:

- Language-level knowledge
- Method-level knowledge

Language-level knowledge expresses the rules of the syntax and semantics of the language used in the specific method, and the rules for meeting various properties of documentation written using the language, such as consistency, satisfiability, and completeness. Thus, the knowledge can be applied to support the automation of documentation construction and verification, and can possibly provide an effective way to *prevent* faults during documentation (e.g., specification, design).

Method-level knowledge records all the knowledge related to the method the ISEE is expected to support, and is usually used to guide human developers in applying the method to develop software systems. Techniques such as help panels, checklists, context-dependent menus, and checking mechanisms are used to prompt the developer in the proper use of the method. Such an ISEE is able to provide context-dependent advice about how to get a software project started, what to do next, what the inputs and outputs are of each step of the method, and how to check the properties of the system under development.

To provide useful knowledge about a specific software development method, the language used in the method must have a formal syntax and semantics, and the rules for developing systems provided by the method must be precise enough to allow for their formal expression in the knowledge base. In this regard, formal engineering methods have obvious advantages over informal methods.

20.3.3 Combination of Both

Since each of the domain-driven and method-driven approaches focuses only on the support for one kind of knowledge, its effectiveness in supporting software development may be limited when it is applied individually, because a development usually needs both domain-based knowledge and method-based knowledge; lacking either of them would affect the productivity and the reliability of software projects.

The combination of both approaches provides greater potential for improving the performance of each kind of ISEE. Since a combined ISEE supports both domain-based and method-based knowledge, it needs to keep both kinds of knowledge consistent in providing guidance and checking documentation. Perhaps an integrated knowledge base, in which each rule reflects a proper combination of domain-based and method-based knowledge, is a better solution to resolve the consistency problem.

20.4 ISEE and Formalization

It has become apparent that knowledge is necessary in order to provide intelligence in an ISEE. However, the most challenging problem in obtaining knowledge is how to easily extract, form, and express the knowledge about the

domain and the software development method, if a realistic ISEE is desired. In our view, *formalization* of both documentation produced during software development and the software development methods are necessary conditions for achieving an ISEE.

The formalized documentation, such as formal specification, has precise syntax and semantics; therefore, there is a high possibility of building a software tool to analyze the documentation. The capability of analysis may lead to the possibility of the presentation of appropriate guidance to the developer and/or of a high degree of automation in specification verification and validation.

The formalization in software development methods means that all the rules for specifications construction, evolution, refinement, transformation, verification, and validation are well-defined. Thus, the rules can be properly incorporated into knowledge about software development methods to provide a method-based intelligent software engineering environment.

In fact, compared with informal languages and methods, formal engineering methods have many more advantages. It is hard to imagine that we can build an efficient and reliable ISEE based on informal documentation and informal development methods, because that would require an incredible amount of time, and intelligence in understanding, analyzing, and manipulating informal languages that have neither well-defined syntax nor formal semantics. On the other hand, with the support of ISEE, the usability and accessibility of formal engineering methods will likely be improved, and the productivity and reliability of the software products developed will likely be enhanced.

20.5 ISEE for SOFL

Since the domain-driven approach to building an ISEE for SOFL needs the involvement of specific domains which are, in general, difficult to describe, this section focuses on the description of the method-driven approach. In other words, we are interested only in the issue of building an ISEE to support the SOFL specification language and method for developing software systems at large.

An ISEE for SOFL is expected to support requirements analysis, specification construction, design, refinement, transformation, rigorous reviews, testing, documentation, system modification, and process management.

20.5.1 Support for Requirements Analysis

The ISEE supports an interactive approach to requirements analysis by following the method for building a specification. Once the ISEE is started, the user, the developer of the desirable software system, of the environment is

requested to input the overall goal, possibly in a natural language like English, and will be guided by the ISEE to develop the informal specification by following the "abstraction and decomposition" principle to define modules in an informal manner. The interaction under the guidance of the ISEE goes on until the input of all the necessary requirements, at an abstract level, terminates.

The ISEE then produces a well-formed informal requirements specification that is consistent with SOFL syntax, and tries to instruct the user to carry out the next step's task: transforming the informal specification into a semi-formal specification. Again, the process is interactive, but the ISEE gives heuristic suggestions in declaring data types, store variables, invariants, data flows, the CDFDs, and in defining processes and functions of modules. In such an interactive way, the user is guided to explore all the possible aspects of the requirements. After the interaction terminates, the ISEE automatically checks the consistency and validity of the semi-formal specification produced, and helps perform a modification of the specification.

20.5.2 Support for Abstract Design

The ISEE takes the semi-formal specification as input and instructs the user to start constructing a design specification accordingly. The top level module is formed automatically and the associated CDFD is drawn. Then, the ISEE takes an interactive approach to guide the user to complete the design specification in a top-down or middle-out manner. During the process of specification construction, the user is usually required to input the information that is impossible for the ISEE to understand, such as new CDFDs and pre and postconditions of processes. What the ISEE does is to raise questions and request necessary information from the user on the basis of its analysis of the consistency, satisfiability, and completeness of the current specification. For example, the ISEE may request from the user the definitions of additional classes and data types, existing given types, and processes that need to produce the open input and/or consume the output data flows of CDFDs in order to complete the design specification. Also, the ISEE assists the user in drawing CDFDs, and automatically creates the associated module outlines to help the user to concentrate on providing inputs for specific components of the module (e.g., pre and postcondition of a process).

20.5.3 Support for Refinement

After an abstract design specification is constructed, the ISEE takes an interleave approach to aiding the user refine the abstract design specification of each process and function, usually in an implicit form, into an explicit specification. To start the refinement process, the ISEE takes initiative in refining the implicit specification of each process. If the specification cannot be refined

completely into an explicit one, the unrefined parts in the implicit specification will be highlighted to signal the user to refine those parts manually. The ISEE is able to accept the user's inputs of the explicit specifications and substitute them for the unrefined parts in the explicit specification generated by the ISEE. This interleave process continues until an explicit specification is completely achieved.

20.5.4 Support for Verification and Validation

The design specifications, both implicit and explicit ones, need to be verified and validated before they are implemented. The ISEE guides the user in conducting rigorous reviews, and in the testing of the specifications for verification and validation. The support includes automatic derivation of various properties from the specifications concerned, automatic construction of review task trees and generation of test cases, and interactive reviewing of the properties or evaluating of the test results. Also, the ISEE issues appropriate advice, whenever necessary, to the user to conduct more but essential reviews or testing. Furthermore, to help in the understanding of design specifications, the ISEE also conducts automatic simulation based on specifications to demonstrate how systems will behave when the programs implementing them are executed.

20.5.5 Support for Transformation

The ISEE provides aids for transformation from design specifications into Java programs in two ways: data transformation and functional transformation. The ISEE usually takes a top-down approach to work on the transformation starting from the top-level module, and then proceeding to its decompositions and to the related classes. For each abstract data type defined in the specification, the ISEE gives the corresponding target data type in the program. For each module, class, process, method, and function, the ISEE usually suggests an outline for the target programs, and requests the user to fill in the contents. Of course, the ISEE tries to do as much as possible in the transformation.

20.5.6 Support for Program Testing

To validate the ultimate program system against the user requirements, the ultimate program produced by transformation of the design specification must be tested under the guidance of the ISEE. Both functional and structural testing are supported. In this process, the test cases generated for testing the design specification can be reused for functional testing of its program. However, the new test cases for structural testing also need to be generated, under the ISEE's guidance, aiming at detecting faults that occurred during the implementation phase. The ISEE supports both interactive and batch testing, and takes care of test case management and program debugging.

20.5.7 Support for System Modification

Modification of documentation is an intrinsic feature of software development; it may be conducted throughout the development process when abstract specifications are evolved or refined into concrete specifications. The ISEE provides several levels of support for documentation modification. When a specification, at any level, is modified, the ISEE highlights automatically all the related parts in the specification to draw attention from the user for possible modifications of the those parts. When an abstract specification is refined into a concrete specification, if any additional modification is made to the concrete specification that leads to the violation of the refinement rules, the corresponding parts of the abstract specification will be marked automatically to indicate the necessity of modification at those parts in order to sustain the refinement rules. Another level of modification may occur when an abstract specification is modified to meet new requirements. In this case, the ISEE will highlight automatically the corresponding parts to be modified in the concrete specification.

20.5.8 Support for Process Management

The software development process is an important element affecting the quality of software products. In the ISEE, software development process is decided by the software environment based on the application domain and the existing knowledge of SOFL software process. The management activities supported may include (1) planning and defining the software process, (2) producing documentation, (3) analyzing project risks, (4) controlling the progress of the software process, and (5) handling exceptional incidents (e.g., over budget or behind schedule). All the activities must be organized in a manner to facilitate systematic development and maintenance of software products.

It is worthy of mention at the end of this chapter that, although the functional features of the ISEE for SOFL described above are not a reality yet, they show the goals for building an ISEE for SOFL in the future. We have so far implemented several prototype tools for SOFL, including a graphical user interface for building specifications and tools to support for specification testing and rigorous reviews, and will continue to make efforts to develop them into a prototype software engineering environment for SOFL. Readers who are interested in this environment and its further development can pay attention to my homepage at *http://wwwcis.k.hosei.ac.jp/~sliu/* for timing information. The current version of the software engineering environment for SOFL is intended to evolve into a more intelligent one in the future.

20.6 Exercises

1. Describe the major differences between a traditional software engineering environment and an intelligent software engineering environment.
2. Give some ideas about building an intelligent office environment by simulating the idea of an intelligent software engineering environment.
3. Explain why it is important that human developers be treated as a "software tool" in an intelligent software engineering environment.
4. Draw a diagram to depict an intelligent software engineering environment for SOFL that provides all the functions presented in Section 20.5. Those functions need to be arranged logically in the diagram to show support for the entire software development process using SOFL.

References

1. Software Engineering, Report on a Conference. NATO Scientific Affairs Division, Garmisch, 1968.
2. Software Engineering Techniques, Report on a Conference. NATO Scientific Affairs Division, Rome, 1969.
3. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
4. K. Arnold and J. Gosling. *The Java Programming Language*. Sun Microsystems, Inc., 1996.
5. J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.
6. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
7. R. Banach. Maximally abstract retrenchments. In S. Liu, J. McDermid, and M. Hinchey, editors, *Proceedings of the Third International Conference on Formal Engineering Methods (ICFEM2000)*, York, UK. IEEE Computer Society Press.
8. D. Bell. *Software Engineering*. Third Edition, Addison-Wesley, 2000.
9. G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, pages 387–405, November 1991.
10. B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
11. B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72, May 1988.
12. W. Brauer, G. Rozenberg, and A. Salomaa. *Petri Nets: An Introduction*. Springer-Verlag, Berlin Heidelberg, 1985.
13. M. Broy and K. Stolen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer-Verlag, 2001.
14. A. Bryant. Structured methodologies and formal notations: developing a framework for synthesis and investigation. In *Proceedings of Z User Workshop*. Oxford 1989, Springer-Verlag, 1991.
15. D. Budgen. *Software Design*. Addison-Wesley, 1994.
16. J. R. Cameron. *JSP and JSD: the Jackson Approach to Software Development*. IEEE Computer Society, 1989.

17. E. M. Clarke, O. Grumber, and D. Peled. *Model Checking*. MIT Press, 2000.
18. L. L. Constantine. *Fundamentals of Object-Oriented Design In UML*. Addison-Wesley, 2000.
19. D. Craigen, S. Gerhart, and T. Ralston. *Industrial Applications of Formal Methods to Model, Design and Analyze Computer Systems: an International Survey*. Noyes Data Corporation, USA, 1995.
20. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.
21. I. S. Daniel Jackson and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proceedings of the International Conference on Software Engineering*, Limerick, Ireland, June 2000. IEEE Computer Society Press.
22. H. M. Deitel and P. J. Deitel. *Java How to Program*. Fifth Edition, Prentice Hall, 2002.
23. T. DeMarco. *Structured Analysis and System Specification*. Yourdon Inc., New York, 1978.
24. E. Dijkstra and C. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
25. J. Dong and B. Mahony. Active objects in TCOZ. In J. Staples, M. Hinchey, and S. Liu, editors, *2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 16–25. IEEE Computer Society Press, December 1998.
26. J. S. Dong and S. Liu. An object semantic model of SOFL. In K. Araki, A. Galloway, and K. Taguchi, editors, *Integrated Formal Methods 1999*, pages 189–208, York, UK, June 28-29 1999. Springer-Verlag.
27. E. Durr and J. Katwijk. VDM++ - a formal specification language for object oriented designs. In *the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'Europe92)*, pages 63–78. Prentice Hall, 1992.
28. M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
29. J. Fitzgerald and P. G. Larsen. *Modelling Systems*. Cambridge University Press, 1998.
30. M. Fowler and K. Scott. *UML Distilled: a Brief Guide to the Standard Object Modeling Language (2nd Edition)*. Addison-Wesley, 2002.
31. K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment: an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proceedings of the First International Conference on Formal Engineering Methods*, pages 170–181. IEEE Computer Society Press, November 12-14, 1997.
32. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2nd edition, 2002.
33. T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.
34. J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.
35. M. Gordon. HOL: a proof generating system for high order logic. In G. BIRTWISTLE and P. SUBRAMANYAM, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, 1988.

36. M. Gordon and A. Pitts. The HOL logic and system. In J. Bowen, editor, *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems series*. Elsevier, 1994.

37. B. S. Gottfried. *Programming with Pascal*. McGraw-Hill, 1994.

38. T. R. L. Group. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice Hall, 1992.

39. J. V. Guttag and J. J. Horning. *LARCH: Language and Tools for Formal Specification*. Springer-Verlag, New York, 1993.

40. D. Hazel, P. Strooper, and O. Traynor. Possum: an animator for the SUM specification language. In *1997 Asia-Pacific Software Engineering Conference and International Computer Science Conference*, pages 42–51. IEEE Computer Society Press, 1997.

41. C. Ho-Stuart and S. Liu. A formal operational semantics for SOFL. In *Proceedings of the 1997 Asia-Pacific Software Engineering Conference*, pages 52–61, Hong Kong, December 1997. IEEE Computer Society Press.

42. C. Hoare. An axiomatic basis of computer programming. *Comm. ACM*, (12):576–580,583, 1969.

43. C. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall Europe, 1998.

44. J. Hoare, J. Dick, D. Neilson, and I. H. Sørensen. Applying the B technologies to CICS. pages 74–84.

45. W. E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, SE-13(10):997–1005, October 1986.

46. http://www.borland.com/jbuilder/.

47. http://www.rational.com/products/rose/index.jsp.

48. D. Jackson. Alloy: a lightweight object Modelling notation. Technical Report 797, MIT Laboratory for Computer Science, Cambridge, MA, February.

49. D. Jackson. Abstract model checking of infinite specifications. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods, Proceedings of Second International Symposium*, pages 519–531, Barcelona, Spain, October 1994. Formal Methods Europe, Lecture Notes in Computer Science 873, Springer-Verlag.

50. M. Jackson. *System Development*. Prentice Hall, 1983.

51. M. Jackson. *Software Requirements and Specifications: a Lexicon of Practice, Principles, and Prejudices*. Addison-Wesley, 1995.

52. M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

53. C. B. Jones. Specification, verification and testing in software development. In *Software Requirement, Specification and Testing*, pages 1–13. Blackwell Scientific, 1985.

54. C. B. Jones. *Systematic Software Development Using VDM*. 1st edition, Prentice Hall, 1986.

55. C. B. Jones. *Systematic Software Development Using VDM*. 2nd edition, Prentice Hall, 1990.

56. P. Juliff. *Program Design*. Prentice Hall, 1990.

57. T. Katayama. A theoretical framework of software evolution. In *Proceedings of International Workshop on Software Evolution (IWPSE98)*, 1998.

58. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. AT&T Bell Laboratories, 1988.

59. N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

60. R. C. Linger and C. J. Trammell. Cleanroom software engineering: theory and practice. In M. G. Hinchey and J. P. Bowen, editors, *Industrial-Strength Formal Methods in Practice*, pages 351–372. Springer-Verlag, 1999.

61. S. Lippman. *C++ Primer*. AT&T Bell Laboratories, 1991.

62. S. Liu. A case study of modeling an ATM using SOFL. Technical report HCIS-2003-01.

63. S. Liu. General Chair's Message. In M. Hinchey and S. Liu, editors, *Proceedings of the First International Conference on Formal Engineering Methods (ICFEM'97)*, page ix, Hiroshima, Japan, November 12-14 1997. IEEE Computer Society Press.

64. S. Liu. An evolution approach for software development using SOFL methodology. In *Proceedings of International Workshop on Principles of Software Evolution*, 1998.

65. S. Liu. Formal methods and intelligent software engineering environments. *Information: An International Journal*, (Vol. 1, No.1):83–102, 1998.

66. S. Liu. Software development by evolution. In *Proceedings of Second International Workshop on Principles of Software Evolution (IWPSE99)*, pages 12–16, Fukuoka City, Japan, July 16-17 1999.

67. S. Liu. Verifying formal specifications using fault tree analysis. In *Proceedings of International Symposium on Principle of Software Evolution*, pages 271–280, Kanazawa, Japan, November 1-2 2000. IEEE Computer Society Press.

68. S. Liu. A rigorous approach to reviewing formal specifications. In *Proceedings of 27th Annual IEEE/NASA International Software Engineering Workshop*, pages 75–81, Greenbelt, Maryland, USA, December 4-6, 2002. IEEE Computer Society Press.

69. S. Liu. Capturing complete and accurate requirements by refinement. In *Proceedings of 8th IEEE International Conference on Engineering of Complex Computer Systems*, pages 57–67, Greenbelt, Maryland, USA, December 2-4, 2002. IEEE Computer Society Press.

70. S. Liu. Developing quality software systems using the SOFL formal engineering method. In *Proceedings of 4th International Conference on Formal Engineering Methods (ICFEM2002), LNCS 2495*, pages 3–19, Shanghai, China, October 21-25, 2002. Springer-Verlag.

71. S. Liu. A formal specification of Shaoying Liu's lab research award policy for students. Technical report HCIS-2003-02, Faculty of Computer and Information Sciences, Hosei University, Koganei-shi, Tokyo, Japan, 2003.

72. S. Liu. Verifying consistency and validity of formal specifications by testing. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, Lecture Notes in Computer Science, pages 896–914, Toulouse, France, September 1999. Springer-Verlag.

73. S. Liu. Evolution: a more practical approach than refinement for software development. In *Proceedings of Third IEEE International Conference on Engineering of Complex Computing Systems*, pages 142–151, Como, Italy, September 8-12, 1997. IEEE Computer Society Press.

74. S. Liu, M. Asuka, K. Komaya, and Y. Nakamura. An approach to specifying and verifying safety-critical systems with practical formal method SOFL. In

Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'98), pages 100–114, Monterey, California, USA, August 10-14, 1998. IEEE Computer Society Press.

75. S. Liu, M. Asuka, K. Komaya, and Y. Nakamura. Applying SOFL to specify a railway crossing controller for industry. In *Proceedings of 1998 IEEE Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98)*, Boca Raton, Florida USA, October 20-23, 1998. IEEE Computer Society Press.

76. S. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: a formal engineering methodology for industrial applications. *IEEE Transactions on Software Engineering*, 24(1):337–344, January 1998. Special Issue on Formal Methods.

77. S. Liu, J. Offutt, M. Ohba, and K. Araki. The SOFL approach: an improved principle for requirements analysis. *Transactions of Information Processing Society of Japan*, 39(6):1973–1989, 1998.

78. S. Liu, M. Shibata, and R. Sat. Applying SOFL to develop a university information system. In *Proceedings of 1999 Asia-Pacific Software Engineering Conference (APSEC'99)*, pages 404–411, Takamatsu, Japan, December 6-10, 1999. IEEE Computer Society Press.

79. S. Liu and J. Woodcock. Supporting rigorous reviews of formal specifications using fault trees. In *Proceedings of Conference on Software: Theory and Practice, 16th World Computer Congress 2000*, pages 459–470, Beijing, China, August 21-25, 2000. Publishing House of Electronics Industry.

80. K. K. M. D. Fraser and V. K. Vaishnavi. Informal and formal requirements specification languages : bridging the gap. *IEEE Transactions on Software Engineering*, 17(5):454–466, May 1991.

81. B. Mahony and J. Dong. Timed communicating object Z. *IEEE Transactions on Software Engineering*, 26(2), February 2000.

82. H. Masuhara and A. Yonezawa. A reflective approach to support software evolution. In *Prooceedings of International Workshop on Principles of Software Evolution (IWPSE'98)*, pages 135–139, Kyoto, Japan, April 1998.

83. C. Mcclure. *Case Is Software Automation*. Prentice Hall, 1989.

84. J. A. McDermid. *Software Engineer's Reference Book*. Butterworth-Heinemann, 1991.

85. S. R. L. Meira and A. L. C. Cavalcanti. Modular object-oriented Z specifications. In C. J. van Rijsbergen, editor, *Workshop on Computing Series*, Lecture Notes in Computer Science, pages 173–192, Oxford, UK, 1990. Springer-Verlag.

86. B. Meyer. *Object-oriented software construction*. Prentice Hall International Series in Computer Science, 1988.

87. E. A. Meyers and J. C. Knight. An improved inspection technique. *Communications of the ACM*, 36(11):50–61, 1993.

88. Michael G. Hinchey and Jonathan P. Bowen (editors). *Industrial-Strength Formal Methods in Practice*. Springer-Verlag, 1999.

89. T. Miller and P. Strooper. Model-based specification animation using test-graphs. In C. George and H. Miao, editors, *Proceedings of 4th International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, pages 192–203. Springer-Verlag, October 2002.

90. C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

91. A. J. Offutt and S. Liu. Generating Test Data from SOFL Specifications. *Journal of Systems and Software*, 49(1):49–62, December 1999.

92. T. J. Ostrand, R. Sigal, and E. J. Weyuker. Design for a tool to manage specification-based testing. In *Proceedings of the Workshop on Software Testing*, pages 41–50, Banff, Alberta, July 1986. IEEE Computer Society Press.

93. M. Page-Jones. *Fundamentals of Object-Oriented Design in UML*. Dorset House Publishing, 2000.

94. D. L. Parnas and D. M. Weiss. Active design reviews: principles and practices. In *Proceedings of the 8th International Conference on Software Engineering*, pages 215–222, August 1985.

95. D. L. Parnas and D. M. Weiss. Active design reviews: principles and practices. *Journal of Systems and Software*, (7):259–265, 1987.

96. N. Plat, J. van Katwijk, and K. Pronk. A case for structured analysis/formal design. In *Proceedings of VDM'91, Lecture Notes in Computer Science*, volume 551, pages 81–105, Berlin, 1991. Springer-Verlag.

97. A. Porter, H. Siy, and L. G. Votta. A review of software inspections. In M. Zelkowitz, editor, *Software Process*, volume 42 of *Advances in Computers*. Academic Press, 1995.

98. R. S. Pressman. *Software Engineering: a Practitioner's Approach*. McGraw-Hill, 2001.

99. H. Qian, E. B. Fernandez, and J. Wu. A combined functional and object-oriented approach to software design. In *Proceedings of First IEEE International Conference on Engineering of Complex Computing Systems*, pages 167–178, Fort. Lauderdale, Florida, November 6-10, 1995. IEEE Computer Society Press.

100. D. Rann, J. Turner, and J. Whitworth. *Z: a Beginner's Guide*. International Thomson Computer Press, 1995.

101. J. M. Rushby and F. von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January, 1993.

102. S. Schneider. *B-Method*. Palgrave, 2001.

103. L. Semmens, R. France, and T. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6), 1992.

104. H. Singh, M. Conrad, and S. Sadeghipour. Test case design based on Z and the classification-tree method. In M. G. Hinchey and S. Liu, editors, *First IEEE International Conference on Formal Engineering Methods*, pages 81–90, Hiroshima, Japan, November 12-14, 1997.

105. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic, 2000.

106. I. Sommerville. *Software Engineering*. Addison-Wesley, 2001.

107. J. Spivey. *The Z Notation: a Reference Manual*. Prentice Hall, 1989.

108. P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

109. T. Tamai and Y. Torimitsu. Software lifetime and its evolution process over generations. In *Proceedings of Conference on Software Maintenance*, pages 63–69. IEEE Computer Society Press, November 1992.

110. The VDM-SL Tool Group. *Users Manual for the IFAD VDM-SL tools*. The Institute of Applied Computer Science, February 1994.

111. H. V. Vliet. *Software Engineering: Principles and Practice*. Wiley, 2000.

112. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, 1999.

113. E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

114. D. A. Wheeler, B. Brykczynski, and R. N. Meeson. *Software Inspection – an Industry Best Practice*. IEEE Computer Society Press, 1996.

115. L. J. White. *Software Testing and Verification*, volume 26 of *Advances in Computers*. Academic Press, 1987.

116. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

117. E. Yourdon. *Modern Structured Analysis*. Prentice Hall, 1989.

A

Syntax of SOFL

This appendix contains a complete definition of the syntax of the SOFL specification language introduced in this book. Although all the keywords, such as **pre**, **post**, **card**, and **len**, are presented in bold font to draw the attention of the readers, they are all given in plain text in the formal definitions of the language. Each definition is given in a context-free grammar in which we use the following conventional symbols:

Meta identifier	non-terminal symbols are written in upper case letters for the first letter and lower case letters for the other parts (possibly including a dash mark)
underline	each terminal symbol is underlined, e.g., <u>process</u>
::=	define symbol
{ }	encloses syntactic items that may occur zero or more times
|	definition separator symbol (with lower precedence than the concatenate symbol)
..	used to describe a range of terminal symbols, e.g., <u>a</u>..<u>z</u>
[ ]	encloses optional syntactic items
( )	encloses the choice of syntactic items

A.1 Specifications

Specification ::= Modules <u>.</u> | Modules; Classes <u>.</u>
Modules   ::= Top-module{; Module}
Classes   ::= Class{; Class}

A.2 Modules

Top-module ::= module SYSTEM_Identifier;
          Module-body end_module
Module   ::= module Identifier [/ Identifier];
          Module-body end_module

   Module-body ::= [Const-declaration;] [Type-declaration;]
               [Var-declaration;] [Inv-definition;]
               [Behavior;]
               Process-function-specifications

   Const-declaration ::= const Identifier = Constant
                    {; Identifier = Constant}
   Type-declaration ::= type Identifier $\equiv$ Type
                   {; Identifier $\equiv$ Type}
   Var-declaration   ::= var Variable: Type
                   {; Variable: Type}
   Inv-definition   ::= inv Predicate-expression
                   {; Predicate-expression}
   Behavior   ::= behav (CDFD_Number
                 | CDFD_Identifier)

   Variable ::= ext Identifier
         | ext #Identifier
         | Identifier

   Process-function-specifications ::= Processes{; Function}

A.3 Processes

Processes   ::= Initialization-process{; Process}
Initialization-process ::= process Init() Process-body
                   end_process
Process   ::= process Identifier([Dataflow-declarations])
                   [Dataflow-declarations]
                   Process-body end_process

   Dataflow-declarations ::= Parameter-declarations[|
                      {Parameter-declarations |}
                      [[| {Parameter-declarations |}]
                      Parameter-declarations]]
           | |Parameter-declarations
                      {|Parameter-declarations}

Process-body ::= [ext Ext-variables] [Precondition]
                 [Postcondition] [Decomposition]
                 [Explicit-specification] [Comment]

Ext-variables   ::= External-variable[: Type]
                       {External-variable[: Type]}
External-variable ::= (rd | wr) Identifier

Precondition ::= pre Predicate-expression
              | pre_Referenced-variable
Postcondition ::= post Predicate-expression
              | post_Referenced-variable

Decomposition ::= decom Identifier_decom

Explicit-specification   ::= explicit Explicit-body
Explicit-body   ::= Local-variable-declaration;
                              Statement
Local-variable-declaration ::= Identifier: Type{; Identifier: Type}
Statement   ::= Sequential-statement
                          | Other-statement
Other-statement   ::= Assignment-statement
                          | Block-statement
                          | Conditional-statements
                          | While-statement
                          | Method-invocation
                          | Multiple-selection-statement

Block-statement   ::= begin Statement end
Assignment-statement   ::= Identifier := Expression
Sequential-statement   ::= Statement{; Statement}
Conditional-statements   ::= if Predicate-expression then Statement
                         | if Predicate-expression
                             then Statement
                             else Statement
While-statement   ::= while Predicate-expression do Statement
Method-invocation   ::= Referenced-method-name
                             ([Arguments])
                         | Method-invocation.Method-invocation
Referenced-method-name ::= Identifier{.Identifier}
                         | this.Identifier
Arguments   ::= Expression{, Expression}

Multiple-selection-statement ::= <u>case</u> Expression <u>of</u> Case-alternatives
[<u>;</u> Default-expression] <u>end_case</u>
Case-alternatives   ::= Case-alternative{<u>;</u> Case-alternative}
Case-alternative   ::= Case-patterns <u>-></u> Statement
Case-patterns   ::= Expression{<u>,</u> Expression}
Default-expression   ::= <u>default</u> <u>-></u> Statement
Comment ::= <u>comment</u> Text

A.4 Functions

Function   ::= <u>function</u> Identifier
                            ([Parameter-declarations])<u>:</u> Type
                            [Precondition] [Postcondition]
                            [<u>==</u> Expression] <u>end_function</u>
Parameter-declarations ::= Identifier{<u>,</u> Identifier}<u>:</u> Type
                         | Parameter-declarations<u>,</u>
                              Parameter-declarations

A.5 Classes

Class ::= <u>class</u> Identifier [<u>/</u> Identifier]<u>;</u>
          Class-body <u>end_class</u>

Class-body ::= [Const-declaration<u>;</u>] [Type-declaration<u>;</u>]
               [Var-declaration<u>;</u>] [Inv-definition<u>;</u>]
               [Behavior<u>;</u>] Method-function-specifications

Method-function-specifications ::= Methods{<u>;</u> Function}
Methods   ::= Constructor{<u>;</u> Method}
Constructor   ::= <u>method</u> <u>Init</u>
                                   ([Parameter-declarations])
                                   Process-body <u>end_method</u>
Method   ::= <u>method</u> Identifier
                                   ([Parameter-declarations])
                                   [Identifier<u>:</u> Type]
                                   Process-body <u>end_method</u>

A.6 Types

```
Type            ::= Type-identifier
                |   Basic-type
                |   Set-type
                |   Sequence-type
                |   Composite-type
                |   Product-type
                |   Map-type
                |   Union-type
                |   Special-type
Type-identifier ::= Identifier{.Identifier}
                |   Class-name
Class-name      ::= Identifier
```

```
   Basic-type            ::= nat0
                        |   nat
                        |   int
                        |   real
                        |   char
                        |   bool
                        |   Enumeration
   Enumeration          ::= {Enumeration-value
                            {, Enumeration-value}}
   Enumeration-value  ::= <String-of-characters>
   String-of-characters ::= Character{Character}
```

```
   Set-type ::= set of Type
```

```
   Sequence-type ::= seq of Type
```

```
   Composite-type ::= composed of Field-list end
   Field-list        ::= Identifier: Type
                            {Identifier: Type}
```

```
   Product-type ::= Type {* Type}
```

```
   Map-type ::= map Type to Type
```

```
   Union-type ::= universal
                |   Type {| Type}
```

```
   Special-type ::= sign
```

A.7 Expressions

Expression ::= Ordinary-expression
      | (Expression)

A.8 Ordinary Expressions

Ordinary-expression ::= Compound-expression
            | Unary-expression
            | Binary-expression
            | Apply-expression
            | Basic-expression
            | Set-expression
            | Sequence-expression
            | Map-expression
            | Composite-expression
            | Product-expression

A.8.1 Compound Expressions

Compound-expression ::= If-expression
            | Let-expression
            | Case-expression

   If-expression ::= if Predicate-expression
            then (Expression | Predicate-expression)
            else (Expression | Predicate-expression)

   Let-expression   ::= let Pattern-definition
                         in (Expression | Predicate-expression)
   Pattern-definition   ::= Pattern-equal-definition-list
                     | Pattern-binding
   Pattern-equal-definition-list ::= Identifier = (Expression |
                        Predicate-expression)
                        {, Identifier = (Expression |
                        Predicate-expression)}
   Pattern-binding   ::= Identifier: (Type | Expression)

```
Case-expression     ::= case (Expression | Predicate-expression) of
                        Case-alternatives
                        [; Default-expression] end_case
Case-alternatives   ::= Case-alternative{; Case-alternative}
Case-alternative    ::= Case-patterns –> (Expression |
                        Predicate-expression)
Case-patterns       ::= (Expression | Predicate-expression)
                        {, (Expression | Predicate-expression)}
Default-expression ::= default –> (Expression | Predicate-expression)
```

A.8.2 Unary Expressions

```
Unary-expression ::= Unary-operator Expression
Unary-operator   ::= -
```

A.8.3 Binary Expressions

```
Binary-expression ::= Expression Binary-operator Expression
Binary-operator   ::= +
                    |   -
                    |   *
                    |   /
                    |   div
                    |   rem
                    |   mod
                    |   **
```

A.8.4 Apply Expressions

```
Apply-expression ::= Method-apply
                   | Function-apply
                   | Composite-apply
                   | Product-apply
                   | Operator-apply

Method-apply ::= Function-apply
               | Function-apply.Simple-variable
               | Function-apply.Method-apply

Function-apply ::= Referenced-variable([Arguments])

Composite-apply   ::= Field-select | Modify-expression
Field-select      ::= Identifier{.Identifier}
Modify-expression ::= modify(Identifier, Modifying-field-list)
Modifying-field-list ::= Identifier –> (Expression | Predicate-expression)
                        {, Identifier –> (Expression |
                        Predicate-expression)}
```

Product-apply   ::= modify(Identifier, Modifying-value-list)
Modifying-value-list ::= Index –> (Expression |
                    Predicate-expression)
                    {, Index –> (Expression |
                    Predicate-expression)}
Index   ::= Number

Operator-apply ::= Operator-name(Arguments)
                | Boolean-type-apply

Operator-name ::= abs
                bound
                | floor
                | get
                | card
                | union
                | inter
                | diff
                | dunion
                | dinter
                | power
                | hd
                | tl
                | len
                | elems
                | inds
                | conc
                | dconc
                | Sequence-name
                | dom
                | rng
                | domrt
                | rngrt
                | domdl
                | rngdl
                | inverse
                | override
                | comp
                | Map-name
Sequence-name   ::= Identifier
Map-name   ::= Identifier
Boolean-type-apply ::= subset(Arguments)
                | psubset(Arguments)

A.8.5 Basic Expressions

Basic-expression ::= Constant
    | Simple-variable
    | Special-keywords
    | Function-apply

A.8.6 Constants

Constant ::= Basic-type-constant
    | Set-type-constant
    | Sequence-type-constant
    | Map-type-constant
    | Composite-type-constant
    | Product-type-constant

Basic-type-constant ::= Sign-type-value
    | Constant-identifier
    | Number
    | Character-value
    | Enumeration-value
Sign-type-value   ::= !
Constant-identifier ::= Identifier{.Identifier}
Number   ::= Integer
    | Real-number
Integer   ::= Negative-number
    | Digits
Real-number   ::= Integer
    | Integer.Digits
Negative-number   ::= - Digits
Digits   ::= Digit{Digit}

Digit ::= 0
    | 1
    | 2
    | 3
    | 4
    | 5
    | 6
    | 7
    | 8
    | 9

Character-value ::= 'Character'

Set-type-constant ::= {[Expression-List]}
Expression-List ::= (Expression | Predicate-expression)
                {, (Expression | Predicate-expression)}

Sequence-type-constant ::= [[Expression-List]]

Map-type-constant ::= { -> }
              | {(Expression | Predicate-expression) ->
                  (Expression | Predicate-expression)
                  {,(Expression | Predicate-expression) ->
                  (Expression | Predicate-expression)}}

Composite-type-constant ::= mk_Referenced-variable
                         (Expression-List)

Product-type-constant ::= mk_Identifier(Expression{, Expression})

A.8.7 Simple Variables

Simple-variable   ::= Identifier | ˜Identifier
                    | Referenced-variable
Referenced-variable ::= Identifier{.Identifier}

A.8.8 Special Keywords

Special-keywords ::= nil
                 | undefined

A.8.9 Set Expressions

Set-expression ::= {Expression |Parameter-declarations &
               Predicate-expression}
             | {Expression |Predicate-expression}
             | Set-type-constant
             | {Integer, ..., Integer}

A.8.10 Sequence Expressions

Sequence-expression ::= [Expression| Parameter-declarations &
                    Predicate-expression]
                  | [Expression | Predicate-expression]
                  | Sequence-type-constant
                  | [Integer, ..., Integer]

A.8.11 Map Expressions

Map-expression ::= {Identifier –> Identifier |
                   [Parameter-declarations &]
                   Predicate-expression}
          | Map-type-constant

A.8.12 Composite Expressions

Composite-expression ::= Composite-type-constant

A.8.13 Product Expressions

Product-expression ::= Product-type-constant

A.9 Predicate Expressions

Predicate-expression ::= true
                  | false
                  | nil
                  | Boolean-variable
                  | Relational-expression
                  | Conjunction
                  | Disjunction
                  | Implication
                  | Equivalence
                  | Negation
                  | Quantified-expression
                  | (Predicate-expression)

A.9.1 Boolean Variables

Boolean-variable ::= Simple-variable
                | Method-apply

A.9.2 Relational Expressions

Relational-expression ::= Expression Relational-operator
                       Expression
                   | Expression Relational-operator
                       Expression Relational-operator
                       Expression
                   | is_Type-identifier((Expression |
                       Predicate-expression))
                   | Boolean-type-apply

Relational-operator ::= $=$
  | $<>$
  | $\leq$
  | $<=$
  | $\geq$
  | $>=$
  | <u>inset</u>
  | <u>notin</u>

A.9.3 Conjunction

Conjunction ::= Predicate-expression <u>and</u>
        Predicate-expression

A.9.4 Disjunction

Disjunction ::= Predicate-expression <u>or</u>
        Predicate-expression

A.9.5 Implication

Implication ::= Predicate-expression $=>$
        Predicate-expression

A.9.6 Equivalence

Equivalence ::= Predicate-expression $<=>$
        Predicate-expression

A.9.7 Negation

Negation ::= <u>not</u> Predicate-expression

A.9.8 Quantified Expressions

Quantified-expression ::= Quantifier-list | Predicate-expression
Quantifier-list   ::= <u>forall</u>[Binding-list]{Quantifier-list}
              | <u>exists</u>[!][Binding-list]{Quantifier-list}

  Binding-list   ::= Identifier-list: (Type | Expression)
            {, Identifier-list: (Type | Expression)}
  Identifier-list ::= Identifier{, Identifier}

A.10 Identifiers

Identifier ::= Letter{(Letter | Digit | _)}
Letter   ::= a..z
          | A..Z

A.11 Character

Character ::= Letter
         | Digit
         | New-line
         | White-space
         | Other-characters

    Other-characters ::= . | , | ; | : | * | ± | - | / | \
                     | | | = | ~ | ( ) | [|] | { | } | @
                     | ^ | ' | & | % | $ | # | " | ! | ?

A.12 Comments

Brief-comment ::= /* Text */
Text   ::= String-of-characters

Index