A Systematic Inspection Approach to Verifying and Validating Formal Specifications based on Specification Animation and Traceability

LI, Mo / 李, 漠

(開始ページ / Start Page)
1
(終了ページ / End Page)
226
(発行年 / Year)
2015-09-15
(学位授与番号 / Degree Number)
32675甲第367号
(学位授与年月日 / Date of Granted)
2015-09-15
(学位名 / Degree Name)
博士(理学)
(学位授与機関 / Degree Grantor)
法政大学 (Hosei University)
(URL)
https://doi.org/10.15002/00012841

仕様アニメーションとトレーサビリティーに基づく形式仕様の検証と実証の系統的な検査アプローチ

A Systematic Inspection Approach to Verifying and Validating Formal Specifications based on Specification Animation and Traceability

by

Mo Li

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

Graduate School of Computer and Information Sciences

HOSEI UNIVERSITY

September 2015

Advisor: Professor Shaoying Liu, Hosei University

Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

First and foremost, I want express my sincere gratitude to my supervisor Prof. Shaoying Liu for his various tremendous supports throughout my research work. He always pays great attentions to the research process of mine and gives me many useful suggestions in doing research. Whenever I encountered problems in my research, I could get timely and useful helps from him. Although he is busy at work, he is always patient in discussing with me, correcting my writings, and leading me to the right direction. He also shares some of his own experience which can benefit my research and daily life. His enthusiasm for research inspired me and all the students in our laboratory, and his encouragement helps me to overcome a lot of difficulties.

I would like to appreciate all the professors in the faculty for their various kind helps during my study. I want to give my special thanks to the professors who have taught me directly in classes. They help me to expand my knowledge in different fields.

I appreciate the student colleagues in the laboratory for their helps in both research and daily life. I also appreciate the students who participated in my experiment for their cooperation.

I want give my thanks to the staff in the administration office for their constant assistance in my campus life.

Finally, I want to thank my family for their understanding and support.

ABSTRACT

The role of requirements analysis in assuring the quality of software products has been well recognized and writing formal specifications has been suggested to be effective in both helping developers understand user's requirements and enhancing the quality of the requirements documentation. However, like the other activities in software development, the construction of formal specifications is usually error-prone in practice. Although many methods have been proposed for error detection, few of them offer systematic and practical approaches to employ the user's requirements in the detection process. How to effectively detect the defects contained in a formal specification before it is delivered for implementation is still a challenge.

In this dissertation, we present a novel inspection approach for formal specification verification and validation. The approach features the combination of a specification animation-based reading technique and a traceability-based checklist. The animation method adopted is called *system functional scenario-based animation method* (SFSBAM) which dynamically presents the operational behaviors of the formal specification by means of "executing" corresponding system scenarios. Each system scenario represents an independent operational behavior and is presented as a sequence of processes. The inspector is guided to read through the formal specification by following its "execution" and required to check the formal specification items of a specific process involved in each step of the "execution" against the informal specification based on a traceability-based checklist. The informal specification documents the user's requirements using a structured natural language and it is the foundation for building the formal specification. The traceability is presented by the relations between the requirement items in the informal specification and their corresponding formalizations in the formal specification. In the checklist, the relations are used to raise specific questions to examine formal specification items for keeping the consistency between the informal and formal specifications.

We also present a prototype tool that supports the entire procedure of our specification inspection approach. Moreover, an experiment has been conducted to evaluate the performance of our inspection approach, and the results indicate that our approach is more effective than the traditional checklisted-based inspection method.

Preface

Nowadays software is deployed in almost all the important systems related to our daily life, such as TV sets, refrigerators, washing machines, automobiles, trains, and airplanes. The quality of the software is therefore crucial to the dependability of the systems. The failure of the software may lead to catastrophic disasters. For this reason, developing reliable software systems has attracted considerable attention from both industry and research communities.

In the traditional software engineering model, requirements analysis is considered as the first phase in the development process. As a result of the phase, a document called requirements specification is usually constructed, which describes the expected functionality of the system and defines the capabilities of the provided software. Since the requirements specification will be used as a foundation for the following development phases, its quality can significantly affect the quality of the final software system.

According to IEEE guide to software requirements specifications [1], a good requirements specification should be unambiguous. However, since the traditional requirements specifications are usually written in natural languages (e.g., Japanese, English), the ambiguities cannot be easily removed. To this end, writing formal specifications has been suggested to be an effective way to help developers understand user's requirements and produce accurate requirements documentation. Formal specifications use mathematically-based formal notations to precisely define the requirements and therefore remove the ambiguity from the specifications. However, like other activities in software development, the construction of formal specifications is usually error-prone in practice. Therefore, detecting errors contained in a formal specification before it is delivered for implementation is very important.

Specification verification and validation is the activity to detect defects and eliminate inconsistencies from the requirements specification for enhancing its quality. On the basis of our study of the literature, we find that there are three main kinds of methods for verifying and validating the formal specifications. One is to formally prove the formal specification. This usually requires that the developer have a strong mathematical background and high skills in manipulating mathematical formulas. The proof process is also usually complex and time-consuming, which can hardly be adopted in practice. Another kind of method is to execute the formal specification by

either translating it into an executable program or using a finite state machine to traverse the state space of the specification. But we should notice that the errors found in the execution can only show the existence of defects but there is no specific instruction to guide the developer to find the defects. The third kind of method is to review the formal specification statically. Using this method, humans are required to read through the specification in order to discover defects. Unfortunately, lack of effective reading techniques limits the effectiveness of such methods in detecting errors. Furthermore, few of existing methods provide precise and systematic guidance for utilizing informal specifications in the defect detection process.

In this dissertation, we propose a novel inspection approach for verifying and validating formal specifications. In the approach, an animation method called system functional scenario-based animation method is adopted as reading technique to guide the inspector to read through the formal specification, and a traceability-based checklist is utilized to instruct the inspector to examine the consistency between the informal and formal specifications. A supporting tool is developed to support the entire inspection process, and an experiment has been conducted to evaluate the performance of our inspection approach.

Major contributions of our inspection approach are briefly introduced below:

1. SFSBAM to support reading in specification inspection

   A *system functional scenario-based animation method* (SFSBAM) is proposed and elaborated. In this method, all of the possible system functional scenarios are first derived from the formal specification. Each system scenario presents an independent operational behavior defined in the specification. In the animation, each system scenario is dynamically presented in a condition data flow diagram. In the inspection, inspectors are guided to read through the formal specification by following the animation process.

2. Traceability-based checklist to support specification inspection

   In our inspection approach, a formal specification is constructed based on an informal specification which documents the user's requirements by using a structured natural language. The traceability between the two specifications is presented by the relations between the requirement items in the informal specification and their corresponding formalizations in the formal specification. In order to formally define the relations, we first formally define the categories of requirement items and formal specification items. Then, traceability

rules between requirement items and formal specification items are formally defined. These relations are used to raise specific questions for each formal specification item for examining the consistency between the informal and formal specifications.

3. A prototype tool to support specifications construction and inspection

   To support the specification inspection process, a prototype tool is developed. The major interesting functionality of the tool includes supporting the construction of both informal and formal specifications, automatically deriving all possible system functional scenarios from the formal specification, carrying out animation as a reading technique to support the specification inspection, and managing all of the related data files.

4. An experiment

   To evaluate the effectiveness of our inspection approach, an experiment is conducted. In the experiment, the subjects are required to inspect the same formal specification by using either our inspection approach or the traditional checklist-based inspection method. The results indicate that our inspection method is more effective than traditional checklist-based inspection method to help the inspector detect defects contained in the formal specification. We analyze the reasons that lead to the results and point out our findings.

Chapter 1 presents the background and the motivation of our research, including the basic terminologies and concepts, the weakness of existing methods, our solution and the contributions of our research**.**

Chapter 2 introduces the SOFL informal and formal specifications, which will be used as the target specifications for illustrating the main principles of our research.

Chapter 3 first gives an overview of our inspection approach and related techniques, and then introduces the background of the case study that will be used to explain the methods and techniques in the following chapters.

Chapter 4 explains the detailed techniques of our system functional scenario-based animation method. The formal definition of the system scenario is given and the algorithm used to automatically extract possible system scenarios is explained. We describe the animation procedure and use examples to illustrate how to carry out the animation.

Chapter 5 describes the traceability between SOFL informal and formal specifications. The categories of requirement items and formal specification items are first formally defined, and the traceability rules for building trace links between requirement items and corresponding formal specification specification items are then discussed. The trace links will be used to construct traceability-based checklist for specification inspection.

Chapter 6 demonstrates how the specification inspection is carried out. We first discuss the four aspects that need to be checked for each formal specification item. Then the structure of a traceability-based checklist is introduced. We discuss the feedback to the questions on the checklist and use a case study to demonstrate the entire inspection process.

Chapter 7 elaborates on the prototype tool that supports the inspection for SOFL formal specification. We explain the design of the tool and introduce its major functions.

Chapter 8 reports an experiment conducted to evaluate the effectiveness of our inspection approach. We first point out the purpose of the experiment, and then introduce the experiment settings. We analyze the experiment results and make several conclusions based on the analysis. We also present several important findings based on our analysis of the experiment result.

Chapter 9 summarizes the research work related to our approach from three aspects: animation, inspection, and traceability. The novelty of our inspection approach is illustrated through the comparison with these related work.

Chapter 10 is the last chapter of this dissertation, which gives the conclusion of the research work presented in the dissertation and points out the future research directions.

Chapter 1

Introduction

Software has become an important part of our modern life style. The smart phone we use to communicate with others works on software; the personal computer we use to finish our daily job works on software; the automobile we use to transfer to different places is controlled by software. From the international company to single person, we all benefit from the efficiency and convenience brought by the usage of software. Since more and more services rely on software, the failure of software system development may lead to great loses [2]. For those safety-critical systems, like control system of powerstation, public transportation, etc., software failures cause not only the loss of money, but also the people's life [3] [4]. How to develop a reliable software system is considered as a critical problem in software engineering.

Software engineering is the study of how to develop and maintain a software system in a systematic approach [5]. Many models have been proposed to manage the software development process. One of the well known models is *waterfall model* [6], that separates the entire software development life-cycle into five phases. As indicated by this model, the first phase in the software development is the *requirements analysis phase*. The major task of this phase is to collect and document user's requirements. It is the most critical phase since it lays the foundation for the subsequent phases, including *design*, *implementation*, *testing*, and *maintenance*. The artifact of the requirements analysis phase is usually a software documentation called *requirements specification*, which describes the expected functions of the system and defines the capabilities of the provided software [7]. Whether a requirements specification is well written can significantly affect the quality of the final software system and the success of a software project [8].

Since the requirements specification will be inherited by the following development phases, any defects in the specification would lead to an incorrect software product and the failure of the development project. Verner *et*

al. in [9] declared that the failures of 73% projects in their 70 failed projects sample are caused by the lack of adequate and correct requirements information. An incorrect product or a failure project may cause many losses. These losses include not only the money, but also the time and the human resources. If the development project is hosted by a software company, the failure will also damage the company's reputation. In order to avoid the failures caused by the incorrect requirements, it is usually cost-effective to remove the errors in the requirements specification as earlier as possible. The earlier the errors in the specification can be found, the more efforts and costs can be saved in the following development. Glass states that requirements errors are the most expensive defects to fix during production but the cheapest to fix early in development [10]. Boehm even claimed that it is possible to save up to $100:1$ by finding and fixing requirements problems early rather than late in the development life-cycle. In addition to the savings, detecting errors in early phase also has significant payoffs in improving the reliability, maintainability, and human engineering of the final product [11].

Specification *verification and validation* is the activity to detect errors and remove inconsistencies from the specification for enhancing its quality [11] [12]. Theoretically, verification and validation are two different activities [13]. Verification is to check whether the specification is right, namely complying with some standards or criteria, and validation is the activity to examine whether the specification satisfies the user's requirements. However, these two activities are almost inseparable in practice since a person (analyst, designer, stakeholder, etc.) must use all his knowledge simultaneously to examine a requirements specification, no matter the knowledge is needed for validation or needed for verification.

Several techniques have already been proposed in different literatures for verifying and validating requirements specifications [14] [15] [16]. The frequently used techniques include specification review, interviews, conventional inspection [17], simulation, animation, formal proof, model checking, and prototyping. Some of these techniques can be applied to both *informal* and *formal requirements specifications*, and some can be applied to only formal requirements specifications. By informal requirements specification, we mean the requirements specification written in natural languages (like English, Japanese). The techniques can be used to verify and validate informal specifications contain reviews, interviews, inspection, etc. However, the inherent characteristics of the informal specification restrict the effectiveness and efficiency of the verification and validation techniques from two aspects:

- The ambiguity in the informal specification cannot be easily removed

The ambiguity of the informal specification is usually caused by the usage of natural languages. The words with multiple meanings and the lack of rigorous sentence structures in the natural language may lead to different interpretations of the same specification content. Moreover, the readers with different backgrounds and domain knowledge may interpret the same specification in different ways.

Some techniques do exist to reduce the ambiguity in the informal specification. For example, the interview technique mentioned above requires an interviewer to discuss a specification with the person who write it for identifying potential blind spots, misunderstandings [11]. The result of an interview is a requirements specification in which the interviewer and the originator have the same understanding. But, the interview cannot guarantee other specification users (developer, tester, operator, maintainer) interpret the specification in the same way as the interviewer and the originator.

- The lack of automated tool support affects the efficiency of verification and validation

Whether automated tool support can be provided affects the efficiency of verification and validation significantly. The tool support can be either fully automated or partly automated. In order to achieve full automation, two conditions must be satisfied. One is that the procedure of verification and validation can be automatically carried out. And the other is that the necessary information used in verification and validation can be automatically extracted from the requirements specification. Whether the procedure can be automatically performed depends on the technique itself. If the procedure of a verification and validation technique cannot be automatically performed, the support to this technique can only be partly automated. Note that the validation activity is to check whether the specification satisfies the user's requirements. Even the technical procedure can be fully automated, the final decision must be made by human.

For example, the specification review requires someone other than the originator, called reviewer, to read through the specification for identifying potential problems. Since the reviewer usually has different point of views, it is possible to find blind spots or misconceptions that the specification developer might have made. In general, the reading process must be done by human and cannot be completed automatically. To efficiently support the reviewing process, the support tool is expected to automatically provide the information that can help the reviewer to make judgement. For instance, when reading the description of a specific function in the specification,

the reviewer need to refer to the related data items or constraints documented in the requirements specification to help him make decisions. If the supporting tool cannot automatically extract such information, the reviewer has to read through the specification trying to find the related information. This will inevitably reduce the efficiency of the tool support and the specification review process. Unfortunately, since the informal requirements specifications lack well defined format and involve ambiguities for interpretations, automated information extraction from informal specifications is almost impossible.

Comparing to informal specifications, formal specifications use mathematically-based formal notations to precisely define the requirements and therefore remove the ambiguity from the specifications [18] [19] [20] [21]. However, since the formal specifications construction process for large systems is usually error-prone, verification and validation are also necessary to enhance its quality. Automated tool support for formal specification verification and validation becomes possible because of the usage of formal notations. In this dissertation, we focus on the technique for verifying and validating the formal specification and its support tool.

1.1 Formal Methods and Formal Engineering Methods

Formal requirements specification is constructed during the procedure of applying formal methods to collect and document user's requirements. In this section, we first introduce the formal methods and related concepts, then we introduce a more practical formal method called *formal engineering method*.

1.1.1 Formal Methods

The concept *formal method* refers to systematic approaches to using mathematical methods for analysis, design, development, and verification of computer and software systems [22]. Different formal methods have been proposed in literatures, such as VDM (Vienna Development Method) [23], Z [24], B-Method [25]. These formal methods have been used to develop different systems [26] [27] [28] [29] to enhance the quality of software products.

The core of formal methods is the formal requirements specification written in mathematically-based formal language. Since the formal language has precise syntax and semantics, the formal specification can precisely define the functions of the expected system without any ambiguity. The first version of formal specification is usually constructed by formalizing the user's requirements. The requirements can be collected through the communications

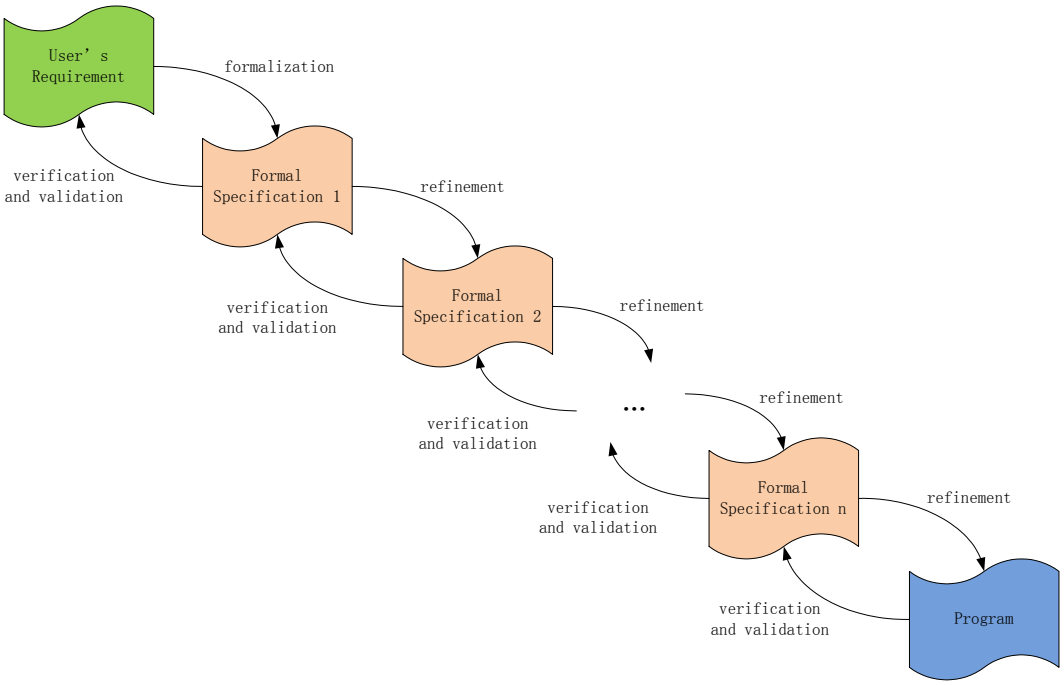


Figure 1.1: The principle of formal methods

with users. Then, the first version of formal specification is evolved through continuous refinement until the final program is implemented. Figure 1.1 shows the principle of applying formal method to software development.

Before the developer can refine the current formal specification into next version, he has to make sure the specification is correct and all user's requirements are appropriately formalized. As indicated in Figure 1.1, this is achieved through verification and validation. Verification is a technique aiming to examine the correctness and internal consistency of the specification, and validation is a technique to check whether the software system formalized in the specification satisfies the user's requirements. Although the effectiveness of validation heavily relies on human decisions, the ideal verification of formal specifications can be rigorous, formal, and fully automated. Moreover, since the formalism is well established in the formal specification, using tool to automatically support the validation process becomes possible.

Although the formal methods provide a theoretically effective solution for developing reliable software systems, the effectiveness of formal methods in realistic systems development is controversial [30] [31] [32]. However, the latest survey conducted by Woodcock and his colleagues in paper [33] indicates that some industrial groups work-

ing in the domain of safety critical systems find formal specifications useful in helping them obtain sufficient understanding of the envisaged system. While recognizing the useful effect of formalization, we find that only mathematically-based notation is unlikely to be widely used in industrial projects in which most of the practitioners do not have a strong mathematical background and their development activities are almost always constrained by limited budget and time [34]. A more practical formal method, known as *Structured Object-oriented Formal Language* (SOFL), has therefore been developed for improvement [35] [36].

1.1.2 Formal Engineering Methods

SOFL is not only a formal specification language, but also a systematic formal approach for software development. This approach is called formal engineering method to distinguish from the traditional formal methods. The characteristics of formal engineering methods include the integration of formal specifications into the modeling process, the combination of formal notations with graphical notations, and automated tool support for writing formal specifications and carrying out inspections and testing. [37] [38] [39] [40] [41] [42] [43].

As a formal specification language, SOFL uses a formalized data flow diagram notation called *condition data flow diagram* (CDFD) to describe the architecture of the system and text-based mechanism called *module* to formally define the components of the CDFD, including *data flows*, *data stores*, and *processes* (or operations in general term), using a formal notation similar to Vienna Development Method - Specification Language (VDM-SL) [44]. Comparing to the traditional formal specifications, like VDM-SL, B-Method, PVS [45], such an architecture-based approach to constructing formal specifications has found to be suitable for abstract design of software systems and helpful in reducing changes in formal specifications [46] [35] [47] [48].

Although the use of the visualized notation CDFD enhances the comprehensibility of SOFL formal specifications for communication, formalizing the user's requirements directly into the formal specification still faces difficulty at the detailed level because of the complexity of the mathematical expressions used in the module. Therefore, a three-step approach for constructing formal specifications is introduced as part of SOFL [36]. In the three-step approach, an informal specification written in a structured natural language is first constructed. The informal specification documents the user's requirements from three aspects: *functions*, *data resources*, and *constraints*. Then, a semi-formal specification is constructed based on the three aspects of the requirements. In the semi-formal

specification, the data resources and constraints are formally defined as necessary data structures and invariants, respectively. The functions in the informal specification are defined as processes. The interface of each process is formally defined, but the functionality of the process is described in a natural language. In the last step, the informal functionality described in the semi-formal specification is formalized for constructing a formal specification.

Unlike the traditional formal methods, the SOFL three-step approach provides a practical and systematic way for creating formal specifications via specification evolution. Since we intend to use the SOFL formal specification to demonstrate our methodology in this dissertation, a detailed introduction is presented in Chapter 2.

1.2 Specification Verification and Validation Techniques

To construct a good formal specification as a fundamental for implementation, verification and validation must be performed to enhance the quality of the specification. Several techniques for formal specification verification and validation have been proposed in literatures. In this section, we summarize some well studied techniques and point out their weakness.

1.2.1 Formal Proof

Formal proof [46], also called *formal verification*, is adopted in the traditional formal methods to guarantee the correctness of formal specifications. The basic idea of formal proof is using logical reasoning to prove the consistency of a formal specification. It provides a fundamental technique for verifying formal specification and is considered as the most rigorous approach for verification. However, the cost-effectiveness and practicality of formal proof are controversial [49] [50].

The proof process is usually complex and time-consuming, and in general cannot be fully automated. Since the modifications of requirements and formal specifications are almost inevitable in the development process, revising formal proofs brings additional costs. Therefore, using formal proof for large scale and complex system is challenging.

The process of formal proof can be supported by theorem provers, such as PVS [45], B-Toolkit [51], and Z/EVES [52]. But using theorem provers usually requires a high level of expertise. The failure of a proof may result from the use of inappropriate proof tactics, inference rules, or existence of errors in the consistency properties.

This means the failure of a proof cannot indicate the incorrectness of the specification, since it can also be caused by incorrect proof process. Moreover, the formal proof is perhaps not necessarily as effective as existing techniques in design and implementation phases. In [49], Hall states that proof is no more a guarantee of correctness than testing, and in many cases far less of one.

Because of its complexity, time-consuming, and the doubt of its cost-effectiveness, formal proof is rarely used in industry [33].

1.2.2 Conventional Inspection

Inspection is a static analysis technique used for the verification and validation of software artifacts, which can be requirements specification, design, or program. The inspection is first proposed in IBM by Fagan in [17], and has been widely used [53] [54] [55] [56] [57] [58]. In an inspection, the target software artifact is first distributed to a team of inspectors that have different perspectives. The inspectors are asked to read the article based on a checklist for identifying possible faults. The checklist contains questions specifying the problems and properties of the target artifact that need to be checked. It serves as a reminder to the inspector to avoid any missing of major defects. As long as the inspectors finish reading the article, the leader of the inspection team will schedule a meeting, called inspection meeting, to gather the team members to discuss their discoveries. The defects that are found in the meeting are documented for further modification.

Many methods and models have been proposed to enhance administrative aspects of the inspection process [59] [60] [61] [62] [63], and some tools have been developed. However, whether an inspection can result in a software artifact with high quality depends on how many potential defects can be found when reading the artifact. In order to improve the effectiveness of the inspection, many methods have been developed to provide reading techniques to help inspector read through the specification [17] [64] [65] [66]. But due to the lack of rigorous and precise definition of the reading process, building effective tool support for reading process is generally difficult.

1.2.3 Animation

Animation is developed as a technique for verifying and validating requirements specifications. It dynamically presents the operational behaviors defined in the specification to give the end users and the field experts with

a chance to interact with the specification and observe its functionality [67]. An animation should provide an intuitive way to the users to monitor the states of a behavior so that they can check whether the specification reflects their original expectation.

To perform formal specification animation effectively, some tools have been built to support animation of different specification languages, such as SOFL Animator [68], ProB [69], and VDMTools [70]. The operational behaviors of a specification are presented in different ways in these tools. For instance, ProB uses a list of invoked functions to describe the behavior and a finite state machine to present the states change of a behavior. SOFL Animator presents the behaviors in Message Sequence Chart (MSC).

In order to dynamically present the defined behaviors, most of these tools require either a translation from a formal specification language to an executable programming language or an interpreter for the executable specification. In this approach, the specification is executed based on some provided input values, and the user can observe the execution and analyze the results. However, there are two issues faced by this approach. One is that the translation or execution of specifications may impose many restrictions on the style of the specifications, and the other is that not all of the specifications can be executed or translated into an executable programming language.

1.2.4 Model Checking

Model checking is a specification verification technique that can be performed automatically [71]. The idea of this technique is to explore all possible states of a specification or model to check whether there is any possibility that the pre-defined properties are violated. For example, UPPAAL [72] is a toolkit for building models and automatically performing model checking. The models in UPPAAL are described by finite state machines rather than text-based specifications. The states and transitions between states in the state machine are formally defined. The user can set invariants to each state, and set guards and actions to each transition. The invariants describe the condition that should be satisfied by the state; the guards restrict the possible state changes by disabling transitions; and the actions change the value of the variables involved in the state. By using the model checker provided by UPPAAL, all possible states of a state machine will be checked against relevant invariants.

The well-known weakness of model checking is the state explosion problem, which is caused by the infinite state space of the target system [73] [74]. The state explosion problem may be solved by abstraction in functions or

restriction on the range of data types. But making the abstraction and restriction requires high-level skills, and whether the refined model can be used as a representative of the original one may not be guaranteed.

1.3 Our Solution: Inspection based on Specification Animation and Traceability

In order to improve the effectiveness and efficiency of formal specification verification and validation, we put forward a new approach called *Inspection based on specification animation and traceability* (IBSAT) that utilizes advantages of the above existing techniques and avoid their weaknesses.

As briefly introduced in the previous section, the inspection is a static analysis technique. The effectiveness of an inspection highly depends on whether the potential errors can be effectively found by the inspector during reading the specification. However, few conventional inspection methods provide detailed reading techniques and precisely defined checklists to guide the inspector reading through and examining the formal specification [75]. In the IBSAT, a new animation based reading technique and a traceability based checklist are proposed.

The animation method adopted in our inspection approach is called *system functional scenario-based animation method* [76]. Comparing to other existing animation methods, this method does not need the translation from formal specification languages to programming languages. The operational behaviors in our animation method are presented as system functional scenarios (or *system scenarios* for short) rather than the execution paths. A system scenario in SOFL formal specification can be formally defined as a sequence of processes or graphically presented as a data flow path in the CDFD. To dynamically demonstrate an operational behavior, every process involved in a system scenario is demonstrated in turn to the inspector. Such step-by-step style can guide the inspector read through the formal specification in a systematic way. The animation performed on the CDFD also provides inspector with an intuitive way for understanding how each part of the formal specification works together. See Chapter 4 for details.

The checklist in our inspection approach is built on the basis of the traceability of specifications [77] [78]. The traceability can be either *internal* or *external*. By internal traceability, we mean the traceability exists within the same level specification. For example, the internal traceability of informal specification describes the relations between different items defined in the informal specification. In contrast with internal traceability, the external traceability is the traceability between the specifications belonging to different levels. For instance, the traceability

between informal and formal specifications is external traceability. It represents the relations between the informal specification items and formal specification items. In order to avoid unnecessary ambiguity, the term *traceability* used in the rest of this dissertation refers to the external traceability.

The traceability is utilized for building checklist because of: 1) using the informal specification as a representative of user's requirements to help the inspector to validate the formal specification; 2) using the informal specification as a reference to help the inspector to verify the formal specification. The questions on the checklist are designed to examine the formal specification from four aspects: *necessity*, *appropriateness*, *correctness*, and *completeness*. These questions guide the inspector to check the correctness of the formal specification and the consistency between the informal and formal specifications. See Chapter 6 for details.

Unlike the conventional inspection, the reading process and checklist are formally defined in our inspection approach. Therefore, using a tool to automatically support the inspection process becomes possible. Along with the proposed inspection approach, we provide a prototype software tool to support the entire process of specifications construction [79], animation [80], and inspection. In addition, this tool also provides a flexible framework to integrate other techniques through a set of well defined interfaces. Some works of other researchers have already been integrated into it.

The major contributions we make in this dissertation are summarized as follows:

1. SFSBAM to support reading in specification inspection

   A *system functional scenario-based animation method* (SFSBAM) is proposed and elaborated. By this method, all of the possible system functional scenarios are first automatically derived from a CDFD and animation for each of the derived system scenario is then carried out as a reading technique to aid a thorough inspection of its consistency and validity. A system functional scenario is a data flow path in the CDFD that defines an independent relationship between the input and output of the CDFD through the formal specifications of the processes involved in the path, see details in Chapter 3.1.1 and Chapter 4.

2. Traceability-based checklist to support specification inspection

   A set of traceability rules between inspection targets and requirement items are formally defined. The definition of traceability rules include the formal definition of user's requirements in the informal specification

and the formal definition of inspection targets in the formal specification. Based on the formally defined traceability rules, a traceability-based checklist is proposed. The questions on the checklist are designed to check the formal specification from four aspects. Several properties are proposed to restrict the contents of formal specification and its relation with informal specification. Moreover, a dependence chain are proposed to guide the inspector check all related inspection targets. See details in Chapter 5 and 6.

3. A prototype tool to support specifications construction and inspection

   A prototype software tool is developed to support both the construction of formal specifications and the inspection of specifications. The major interesting functionality of the tool includes supporting the construction of both CDFDs and modules, automatically maintaining the structural consistency between the CDFD and the module, decomposing high level processes, automatically deriving all possible system functional scenarios from a designated CDFD, carrying out animation as a reading technique to support the inspection of the consistency and validity of each system scenario, and managing all of the related data files. See details in Chapter 7.

4. An experiment to evaluate the performance of our inspection approach

   An experiment is conducted to evaluate the performance of our proposed inspection method. In the experiment, the subjects are divided into different groups based on their experience in SOFL formal specifications. All the groups are required to inspect the same formal specification but use different inspection methods. We compare our inspection method with the traditional checklist-based inspection method, and the results indicate that our inspection method is more effective to help the inspector to detect defects contained in the formal specification. We analyze the reasons that lead to the results and point out our findings. See details in Chapter 8.

Note that the principles of the proposed approach in this dissertation are not only applicable to the SOFL specification language, but also to other model-based formal specification languages, such as the well known VDM-SL and B-Method because they use similar formal notations and mechanism for constructing formal specifications. We choose SOFL as the target formal technique for discussion in this dissertation partly because it has been recognized as a practical technique by academics and practitioners [47] [81] [82] [83] and partly because it shares

many commonalities with existing model-based formal notations as mentioned previously. Thus, the proposed approach can be easily learned and applied by academics and practitioners even with the background of other existing formal methods.

1.4 Summary

In this Chapter, we first discussed why requirements specification is so important to ensuring the quality of the software product. We also discussed the major challenges of informal specification verification and validation, and then briefly introduced formal methods and formal engineering methods, and discussed some well studied formal specification verification and validation techniques. Finally, we proposed a novel inspection approach based on specification animation and specifications traceability.

In the next Chapter, we will introduce the structure of SOFL informal and formal specifications which will be used to demonstrate our inspection approach.

Chapter 2

Brief Introduction of Structured Object-oriented Formal Language

In our inspection method, the formal specification is verified and validated against the informal specification based on the traceability between the two specifications. The construction of traceability between specifications is critically affected by the structure of the specifications. In this chapter, we briefly introduce the fundamental principle of the SOFL formal engineering methods and the structure of SOFL informal and formal specifications, which will be used as target specifications in our approach. For more details of the SOFL, the reader can refer to the *SOFL* book [36] and other related work [81] [82] [83] [84] [85].

SOFL stands for Structured Object-oriented Formal Language and was first proposed in Liu's paper [86]. It is not only a formal language, but also a software modelling approach. It is designed to bridge the formal methods and their applications in real software development. Comparing to other formal methods, the SOFL formal engineering method provides both a comprehensible formal language and a practical three-step method for constructing formal specifications.

As a formal engineering method, SOFL provides rigorous but practical techniques to build formal specification of software system in a three-step evolutionary manner. In this three-step modelling approach, three kinds of specifications with different level of formalization are constructed, namely the *informal*, *semi-formal*, and *formal specifications*.

The *informal specification* is first constructed in a development process. The major task of building informal specification is to discover and collect all desired requirements from the end user. Informal specifications can

generally be specified in any style as long as it can be used to communicate with the user easily. In SOFL modeling approach, it is specified in a structured natural language containing *functions*, *data resources*, and *constraints*

The *semi-formal specification* is refined from the informal specification. It is a more accurate and better structured specification to enhance communications between the user and the designer and to help the designer clarify ambiguities in the informal specification. All the requirements in the informal specifications are encapsulated into system modules in the semi-formal specification. The desired functions in each system module are defined as SOFL processes. In the semi-formal specification, not everything is fully formalized. The data structures and interface of processes are formally defined, but the functionality of each process is informally defined using structured natural languages. Since the semi-formal specification is used to communicate with the end user to clarify the software system, defining the functionality of the system using informal language is reasonable. The formally defined data structures and informally defined functionality are adopted as basis to construct formal specification.

The *formal specification* refines the semi-formal specification according to the feedbacks from the end user and formalizes the informal parts of the semi-formal specification. The expected function of each SOFL process is formally defined by mathematically-based notations in a manner of pre- and post-conditions. It is intended to precisely and accurately define the functionality and architecture of the software system.

Since the traceability used in our inspection method is constructed between SOFL informal and formal specifications, we introduce the structure of these two kinds of specifications as follows.

2.1 Informal Specification

As shown in Figure 2.1, three sections of requirements must be documented in the informal specification. The first section, "*Functions*", is a collection of desired functions. Each function describes an expected behavior to be implemented in the system under development. A function can be decomposed into several low level functions and all functions are presented hierarchically. Each function in the informal specification is assigned with an unique identifier which consists of a capital character "*F*" and the number in front of the function. For example, the first function in the informal specification shown in Figure 2.1 is "*Withdraw*", and its identifier is "*F1.1*". We will use this identifier to indicate the function "*Withdraw*" in the rest of this dissertation. The identifiers of other

requirement items in the informal specification have the same structure.

Another section in the informal specification is the "*Data Resources*", which works like database to supply necessary data to the functions listed in the "*Functions*" section. The data items listed under "*Data Resource*" can be accessed or updated by more than one function. The identifiers of the functions that have the potential to access a specific data resource are listed in a pair of brackets after the description of the data resource. For instance, the only data resource *D2.1* will be accessed or updated by function *F1.1.2*, *F1.1.4*, *F1.2.2*, and *F1.2.3* as shown in Figure 2.1.

The third section of informal specification is "*Constraints*". It describes a collection of constraints either on the captured functions or the data resources. A constraint shows a restriction condition that usually prevents the system from providing unnecessary functions; it can be used to document nonfunctional requirements, such as business policy, safety, or security. The identifiers of the functions or data resources that have to comply with a specific constraint are listed after the description of the constraint. For example, the function *F1.1.3* is listed after constraint *C3.4*, that means the functionality described in *F1.1.3* must comply with the *C3.4*. Namely, the "*Receive withdraw amount*" function should not accept a withdraw amount that is lager than 10,000.

2.2 Formal Specification

A SOFL formal specification consists of a hierarchy of *CDFDs* and the associated hierarchy of *modules*. Each module in a SOFL specification is a structure, which contains necessary *constant declarations*, *type declarations*, *state variable declarations*, *invariant definitions*, and a collection of *process* specifications. For example, one module named "SYSTEM_ATM" is shown in Figure 2.2 and its associated CDFD is displayed in Figure 2.3. This module contains one type declaration, one state variable, three invariants, and four processes.

Theoretically, each desired function in the informal specification is realized by a *process* specification; each data resource is represented by a *state variable* with a corresponding *type declaration*; and each constraint is mapped to either part of a *process* specification or to an *invariant*, which is a property that must be sustained by all related processes throughout the entire specification. Each *process* in a module of SOFL formal specification presents an independent operation that produces output data based on the input data it receives. It can access and update the data of state variables defined in the same module, which usually realize the data resources described in the

1 Functions
1.1 Withdraw
1.1.1 Receive command
1.1.2 Check password
1.1.3 Receive withdraw amount
1.1.4 Pay cash
1.2 Check balance
1.2.1 Receive command
1.2.2 Check password
1.2.3 Show balance
2 Data Resources
2.1 Bank account database (F1.1.2, F1.1.4, F1.2.2, F1.2.3)
3 Constraints
3.1 Only two commands can be received "withdraw" and "showbalance" (F1.1.1, F1.2.1)
3.2 ID No. of each account should be 4 digital number (F1.1.2, F1.2.2, D2.1)
3.3 Password of each account should be 4 digital number (F1.1.2, F1.2.2, D2.1)
3.4 Maximum amount of withdraw is 10,000 (F1.1.3)

Figure 2.1: The informal specification of a simplified ATM software

informal specification.

For each module, there is a corresponding formal graphical notation called *Condition Data Flow Diagram* (CDFD). The CDFD uses visual notation to intuitively and comprehensibly express the structure of a module, namely the relationships between different *processes* contained in the module. A process in a CDFD is represented by a rectangle box with its name in the center. Different processes are connected by *data flows* to present the potential functions of a formal module. In addition to the data flows, a process can access to the data in a *data store*. A data store is the graphical representation of a state variable declared in the associate module. The relation between a process and a data store in a CDFD is consistent with the relation between the process and the corresponding state variable in the associated module.

The CDFD is both a formal and intuitive notation suitable for describing system structure. Figure 2.3 shows the associated CDFD of the module displayed in Figure 2.2, four processes "Receive_Command", "Check_Password", "Withdraw", "Show_Balance", and a data store "Account_file" are included.

As shown in Figure 2.2, a process in SOFL formal specification consists of five parts: *name*, *input variable list*,

```
module SYSTEM_ATM;

type
Account = composed of
        id: string
        name: string
        password: string
        balance: real
        available_amount: real
       end;

var
ext #Account_file: set of Account;

inv
forall[a: Account] | len(a.id) = 4;
forall[a: Account] | len(a.password) = 4;
forall[a, b: Account] | a <> b  a.id <> b.id;

process Receive_Command(withdraw_comm: string | balance_comm: string)
                                                                    sel: bool
pre true
post withdraw_comm = “withdraw” and sel = true
      or
     balance_comm = “balance” and sel = false
end_process;

process Check_Password(id: string, sel: bool, pass: string)
                acc1: Account | err1: string | acc2: Account
ext    rd #Account_file
pre    true
post   (exists![x: Account_file] | ((x.id = id and x.password = pass) and
       (sel = true and acc1 = x or sel = false and acc2 = x)))
       or
       not(exists![x: Account_file] | (x.id = id and x.password = pass)) and
         err1 = "Reenter your password or insert the correct card"
end_process

process Withdraw(amount: nat0, acc1: Account) cash: nat0 | err2: string
pre...
post...
end_process

process Show_Balance(acc2: Account) balance: nat0
pre...
post...
end_process
end_module
```

Figure 2.2: The formal specification of module “SYSTEM_ATM”

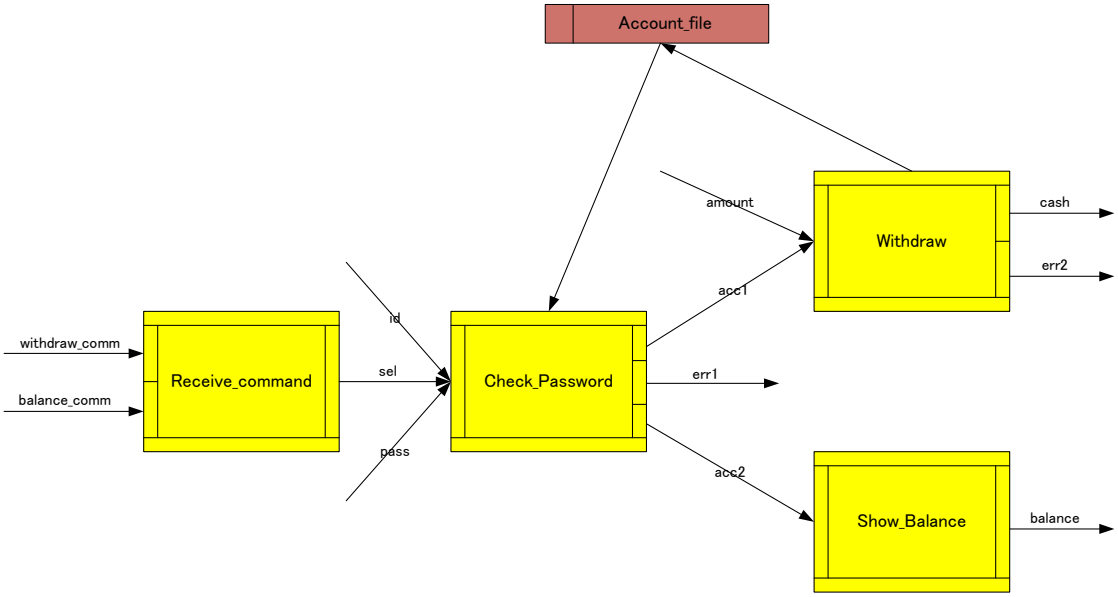


Figure 2.3: The CDFD of a simplified ATM software

output variable list, *pre-condition* and *post-condition*. The name of a process is an identifier. The input variable declarations are put in parenthesis. In some process, the input variable list is divided by vertical bars into several *input ports*. Each input port contains some input variables and is exclusive to each other. For example, the process "Receive_Command" in Figure 2.2 contains two input variables "withdraw_comm" and "balance_comm". These two input variables being separated by a vertical bar indicates that the process "Receive_Command" cannot receive them in the same time. The process can receive either input variable "withdraw_comm" or input variable "balance_comm". The input variable list is followed by the output variable list. It is defined in the same manner as input variable list. The vertical bars can divide the output variable list into *output ports*.

The pre- and post-conditions formally define the semantics of the process using mathematically-based notations. The semantics of a process is interpreted as follows: when one of the input ports of the process is activited, which means that the values of the input variables included in the activited port satisfy the pre-condition, the process will be executed. As a result of the execution, one of the output ports is made avaliable, which means that the values of its output variables satisfying the post-condition are produced based on the input values.

In the CDFD, the input and output ports of a process is denoted by the narrow rectangles on the left and right side of a process, respectively. They are used to receive and send out input and output *data flows*, which are the

counterparts of input and output variables in the CDFD. When one of the input ports of a process receives all the necessary input data flows, the process can be activated and executed to produce corresponding output data flows.

In the SOFL formal specification, the processes contained in the same module can contact with each other via input and output variables. A process can also be decomposed into a lower level module, which contains a group of lower level processes. This decomposition can be continued until the designer considers that the process no longer needs to be decomposed. Generally, the type declarations, state variable declarations, and invariant definitions are used inside the module that defines them. But, if a process is decomposed into a new lower level module, the types, state variables, and invariants specified in the module containing the process can be used directly in the new lower level module.

2.3 Summary

In this Chapter, we briefly introduced the structure of SOFL informal and formal specifications. In the next Chapter, we will give an overview of our inspection approach, and the background of the example used in this dissertation will also be introduced.

Chapter 3

Overview of the Inspection Based on Specification Animation and Traceability

In this chapter, we first introduce the overview of the IBSAT (*I*nspection *B*ased on *S*pecification *A*nimation and *T*raceability) based on the SOFL informal and formal specifications. Then, the background of the example used in this dissertation for demonstrating our inspection approach will be introduced.

3.1 The Outline of IBSAT

The inspection approach proposed in this dissertation consists of two parts: an animation-based reading technique and a traceability-based checklist. The reading technique provides a systematic procedure to guide the inspector reading through the formal specification without missing any important contents, and the traceability-based questions remind the inspector what should be checked. In this section, we briefly introduce these parts as follows.

3.1.1 System Functional Scenario-based Specification Animation

The reading technique plays an important role in an inspection method. Inspection of formal specifications is a static analysis technique and it usually requires the inspector to read through the specification for error detection. An effective reading technique can guide the inspector to read the specification in a systematic manner. It helps the inspector understand the specification and inspect all contents concerned. In our inspection method, we adopt formal specification animation as a reading technique. By specification animation we mean a dynamic visualized demonstration of the system behaviors defined in the specification by means of showing the relation between the

input and output of the system. To ensure that the animation can effectively and efficiently assist the user and the designer to validate the specification against the corresponding informal requirements, utilizing specification animation as a reading technique to facilitate inspection of the specification has found to be beneficial (see Chapter 8).

The animation approach used in the inspection method is called *system functional scenarios-based animation method* (SFSBAM), as mentioned before. The first step of an animation is to derive all possible *system functional scenarios*. Each system functional scenario is intended to describe an individual behavior of the system in terms of the input and output relation, which is similar to a use case in the UML use case diagram [87]. In SOFL formal specification, a system scenario can be formally defined as a sequence of processes and graphically presented as a data flow path in the CDFD. The second step of the animation is to select an appropriate test case (or animation case) and an expected output for animation. The selection process can be either manual or automatic. Usually, the animation case generated automatically cannot present the most concerns of users. The third step in the animation is to use the selected test case and expected output to "execute" the system scenario by "executing" every process involved in the system scenario in turn.

There are two major advantages by adopting system scenario-based animation as a reading technique. The first advantage is that the animation can help the inspector understand the formal specification. Firstly, the animation is performed on the CDFD to provide inspector with an intuitive way for understanding how all of the parts of the formal specification work together. Secondly, the selected test cases represent the states of system behaviors. The inspector can observe the behaviors by monitoring the states to understand the functionality of the formal specification. The other advantage is that animation can guide the inspector to check all important formal specification items. Since the system scenarios represent system behaviors, all related formal specification items will be finally integrated together to perform system scenarios. By inspecting system scenarios, all related formal specification items will be inspected correspondingly.

Specifically, in each step of an animation, a specific process is "executed" during which the inspector is guided by a checklist to read and examine the specification of the process. All the contents of the process specification, including its name, input variables, output variables, pre- and post-conditions, are examined. In the meantime, other formal specification items related to the process (such as type declarations, invariants, etc.) are also inspected.

By inspecting the possible system scenarios, all formal specification items that contribute to the functionality of system defined in the specification are checked. For other formal specification items that have not been inspected, the inspector should consider whether they are defined incorrectly or their formalization is unnecessary.

The details of the system scenario-based animation method is introduced in Chapter 4 and how to use animation as reading technique to carry out an inspection is demonstrated in Chapter 6.

3.1.2 Traceability-based Checklist

In an inspection, the inspector verifies and validates the formal specification by answering the well prepared questions on a checklist. In order to help performing an efficient inspection, two problems must be addressed. One is what to ask, and the other is what kind of information should be provided to support the inspector to answer the questions. To address the first problem, we form questions for the checklist by taking the following four important aspects of the formal specification into account: *necessity*, *appropriateness*, *correctness*, and *completeness*. Several questions are prepared for each aspect, and these must be answered to ensure that the user's requirements are properly formalized in the formal specification.

To handle the second problem, we use the traceability between informal and formal specifications to provide necessary information to help the inspector answer the questions. The traceability presents the connection between the two specifications. When answering the questions about a specific formal specification item, the inspector can trace back to the original requirement of this item by using the traceability, and the requirement can help the inspector to make better judgements to answer the questions.

The traceability between informal and formal specifications is constructed based on some *traceability rules*. In order to formally define the traceability rules, we first formally define the informal specification as a group of *requirement items* and the formal specification as a group of *inspection targets*. An inspection target is an independent unit in the formal specification that needs to be inspected. It may contain one or more formal specification items. The traceability rules describe how a *trace link* can be constructed between an inspection target and a requirement item, which is the original requirement of the inspection target.

The requirement items can be either explicit or implicit. The explicit requirement items are the requirements that are explicitly defined in the informal specification, including functions, data resources, and constraints. The

implicit requirement items are described by the relation between different explicit requirement items. Although the implicit requirements are not defined explicitly in the informal specification, they still need to be treated as user's requirements and formalized in the formal specification.

Similarly, the inspection targets are also divided into explicit and implicit targets. The explicit inspection targets are the formal specification items that explicitly defined in the specification, containing type declarations, state variable declarations, invariants, processes, etc. An implicit inspection target of formal specification is the relation between two different explicit inspection targets. For example, a variable must be declared with a type, the relation between variable and its type is considered as an inspection target and should be inspected.

When an inspection target is under inspected, the inspector needs answer the questions prepared for the current inspection target. The trace link guides the inspector to find the original requirement of the inspection target. The original requirement provides necessary information to the inspector to answer the related questions. However, before the trace links can be used to trace the inspection targets back to their original requirements, the links have to be constructed first. In our approach, the trace links are constructed manually. The construction process will force the inspector to uncover the essence of the inspection target and help the inspector understand the corresponding formal specification items. As long as the trace links are built, the linked requirement items will serve as background to help the inspector answer other questions on the checklist.

3.1.3 Formal Specification Inspection

To inspect a formal specification, the inspector reads through the formal specification by following the system scenario-based animation process. In each step of the animation, one specific process in a SOFL module is "executed". The inspector should check all the formal specification items that are involved in the "execution" of the process. The involved formal specification items and their relations are formalized as inspection targets in our inspection method and inspected against the informal specification. The traceability-based checklist reminds the inspector what to check about an inspection target and where to find necessary information. By answering the questions on the checklist, the inspector verifies and validates the formal specification on the basis of his experience and skills.

Figure 3.1 illustrates the inspection process. The CDFD shown in Figure 3.1 presents a system functional

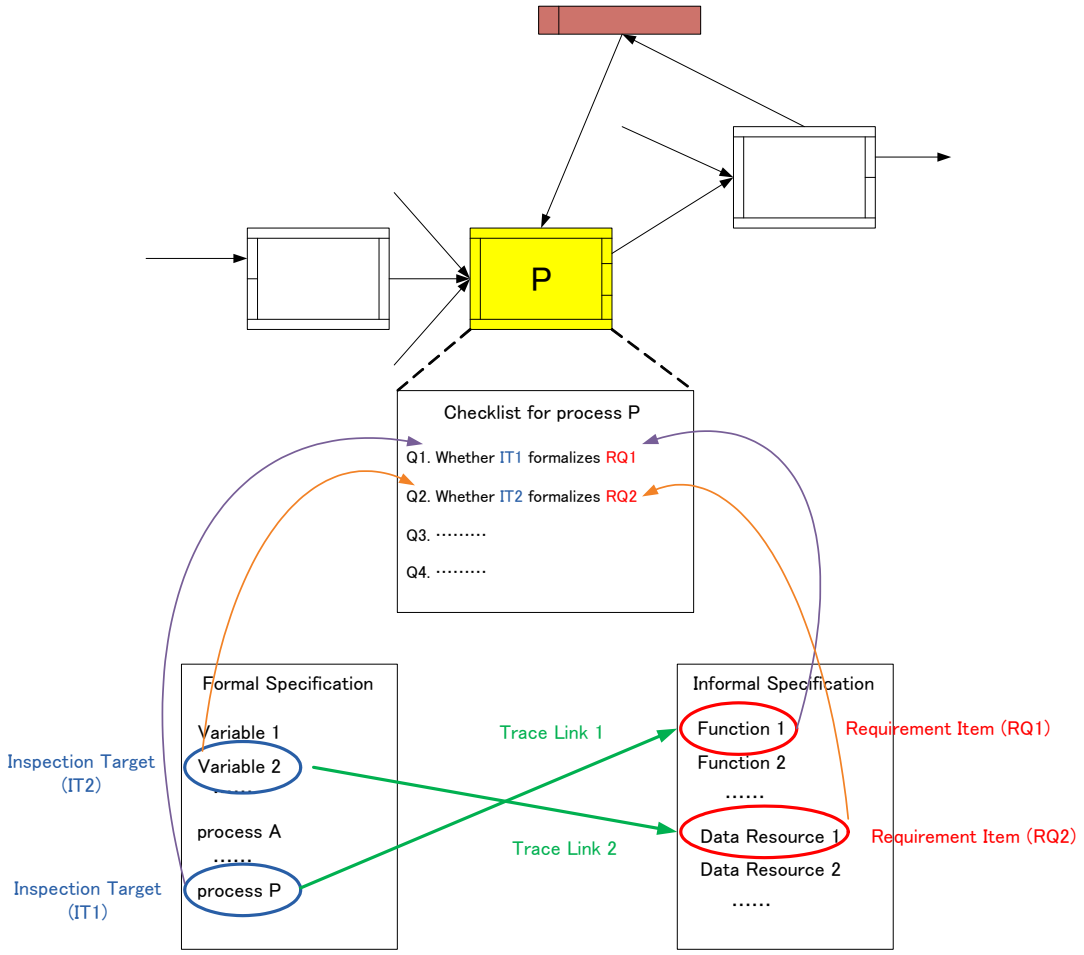


Figure 3.1: The relations among basic concepts of our inspection approach

scenario and the data flows indicate the "execution" order of the processes involved in the system functional scenario. The inspector checks each process in turn by following the "execution" order. As shown in Figure 3.1, the second process "P" in the system functional scenario is under inspection. A traceability-based checklist is provided to the inspector for examining the process.

The questions on the checklist are raised based on the trace links between inspection targets and requirement items. As shown in Figure 3.1, the inspection targets are the items defined in the formal specification and the requirement items are the items described in the informal specification. The trace links are the bridges that connect related inspection targets and requirement items.

To inspect the process "P", all the inspection targets that are related to process "P" need to be checked. The

questions on the checklist are formed to instruct the inspector to examine each inspection target. To raise specific questions, both the inspection targets and associated requirement items are used for questions construction. For example, the inspection target "IT1" in Figure 3.1 is the process specification of "P". The trace link "Trace Link 1" connect the "IT1" with the requirement item "RQ1", which is a function described in the informal specification. To inspect "IT1", requirement item "RQ1" is used for constructing question "Q1" on the checklist. The question asks "Whether IT1 formalizes RQ1?". Similarly, the inspection target "IT2" and requirement "RQ2", which are connected by "Trace Link 2", are used to form the second question on the checklist.

In order to facilitate the inspector to do the inspection, a support tool is developed. This tool can support the entire process displayed in Figure 3.1 including constructing informal and formal specifications, decomposing formal specification into system scenarios, animating system scenarios, and constructing checklist. The inspector can focus on answering the questions and making judgements. The details of this support tool can be found in Chapter 7.

3.2 Background of the Case Study

Before explain the details of our animation-based inspection method, we first briefly introduce an example. This example will be used to demonstrate the concepts and technique details in the following chapters.

The example we use is a simplified ATM system. The system contains only two major functions: withdraw and show balance. The description of these two functions is as follows:

- **Withdraw**

  The ATM system should first receive the "*withdraw*" command. Then, the user should be asked to provide the ID and password of his bank account. If the provided bank account exists in bank's database and the provide ID and password match each other, the user can withdraw money from his or her bank account. However, the amount that can be withdrawn cannot be more than the balance of provided bank account. Meanwhile, the bank requires that the amount that can be withdrawn cannot be more than 10,000 JPY each time. If the user provides an amount that cannot be withdrawn, an error message should be displayed.

- **Show balance**

The ATM system should first receive the "*show balance*" command. Then, the user should be asked to provide the ID and password of his bank account. If the provided bank account exists in bank's database and the provide ID and password match each other, the user can check the balance of the provided bank account, and the balance will be displayed.

The informal and formal specifications of this simplified ATM system are shown in Figure 2.1 and Figure 2.2, respectively. The corresponding CDFD is shown in Figure 2.3. In the informal specification, the two major functions are decomposed into several sub-functions. For example, the "withdraw" function consists of four sub-functions: "Receive command", "Check password", "Receive withdraw amount", and "Pay cash". The only data resource "Bank account database" record all the bank accounts information. Four constraints restrict both the functions and data resource.

The formal specification contains four processes that formalize the functions in the informal specification. Note that the formal specification is not simple formalization of the informal specification. It contains developers' design of the software system. The developer may combine different functions in the informal specification into one process or decompose a function into several processes. The corresponding CDFD depicts the relations of these four processes. In addition to the processes, the formal specification includes one type declaration, one state variable declaration, and three invariants. All these formal specification items will be inspected for verification and validation.

This simplified ATM system is designed for demonstration purpose only. Although it is not a comprehensive system, it is enough for illustrating our approach. A more rigorous evaluation of our inspection approach can be found in Chapter 8.

3.3 Summary

In this Chapter, we explained the outline of our inspection approach and briefly introduced the relevant techniques and concepts underlying the inspection approach. The example used in this dissertation to demonstrate our inspection approach was also introduced. Starting from the next Chapter, we will explain the details of our inspection approach. Each technique will be illustrated in turn. The technique introduced in the next Chapter is the system functional scenario-based animation method.

Chapter 4

Formal Specification Animation

Similar to other animation methods introduced in Chapter 1, our *system functional scenario-based animation method* can be adopted independently to validate formal specifications. Moreover, this animation method can be adopted as a reading technique to support formal specification inspection. In this chapter, we explain our system functional scenario-based animation method in detail.

As introduced previously, a formal specification is first decomposed into a group of exclusive system functional scenarios in our animation method. In order to further explain the details of this animation method, we first formally define system functional scenarios. The following definition indicates how a system scenario is expressed.

Definition 1 *A system functional scenario, or system scenario for short, of a CDFD is a data flow path denoted by the corresponding sequence of processes, $d_i[P_1, P_2, ..., P_n]d_o$, where d_i is the set of input variables of the scenario, d_o is the set of output variables, and each P_i ($i \in \{1, 2, ..., n\}$) is a process, P_1 and P_n being the staring process and the terminating process, respectively, where a starting process must only be activated by its one-end open input data flows and a terminating process must only produce one-end open output data flows.*

The *system scenario* $d_i[P_1, P_2, ..., P_n]d_o$ defines an independent behavior that transforms input data item d_i into the output data item d_o through a sequential execution of processes $P_1, P_2, ..., P_n$. Such a scenario can be perceived as a use case from the user's point of view, defining one pattern of using the system. For instance, from the formal specification and corresponding CDFD of a simplified Automated Teller Machine (ATM) in Figure 2.2 and Figure 2.3, the following five system scenarios can be derived:

- $\{withdraw\_comm\}[Receive\_Command,\ Check\_Password,\ Withdraw]\{cash\}$
- $\{withdraw\_comm\}[Receive\_Command,\ Check\_Password,\ Withdraw]\{err2\}$

- {*withdraw_ comm*}[*Receive_ Command, Check_ Password*]{*err1*}
- {*balance*}[*Receive_ Command, Check_ Password*]{*err1*}
- {*balance*}[*Receive_ Command, Check_ Password, Show_ Balance*]{*balance*}

For simplicity, we omit all of the intermediate data flows, such as "*id*", "*pass*", "*acc1*", "*acc2*", and "*amount*". The first scenario shows that given the input "*withdraw_ comm*", *the three processes*, which are *"Receive_ Command", "Check_ Password", and "Withdraw",* are executed in turn. As a result, the output "*cash*" is produced. The other system scenarios can be interpreted similarly.

In order to animate the entire formal specification, we require that each system scenario be animated. A system scenario is animated by means of "executing" each process involved in the scenario in turn. The "execution" of a process is presented by its input and output data, which can be collected from the user or generated automatically.

In the rest of this chapter, we first introduce how to derive all possible system functional scenarios automatically, and then illustrate how to collect appropriate data and animate a system scenario.

4.1 CDFD Decomposition

The first step of our animation approach is to derive all possible system scenarios. Since each system scenario is expressed as a sequence of processes, deriving all possible system scenarios can be realized as finding all possible sequences of processes. As introduced in Chapter 2, the CDFD is designed to graphically describe the architecture of the module and the connection of processes, therefore, our animation method derives system scenarios from the CDFD.

In order to automatically generate all possible scenarios based on the topology of CDFD, the CDFD has to be decomposed first. Since the topology of CDFD is composed of data flows (edges) and processes (nodes), the decomposition of CDFD can be realized by decomposing every process in the CDFD. The processes must be decomposed because their semantics is much more complex than the semantics of nodes in a normal graph. The semantics of processes must be considered in the derivation of system scenarios. Figure 4.1 illustrates the decomposition of a single process and explains the reason that it must be decomposed.

In SOFL formal specification, each process may have more than one input port and the input ports are exclusive

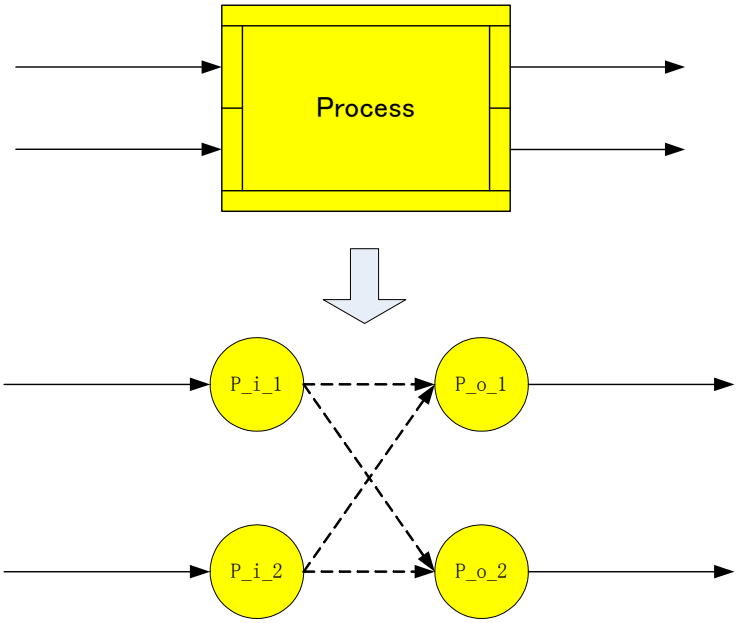


Figure 4.1: The decomposition of a single process

to each other. This means in each time of execution of a process, only one of the input ports of the process can be activated. Similarly, only one of the output ports of the process can be activated in each time. For example, the process in Figure 4.1 has two input ports and two output ports. Theoretically, there are four possible combinations between these ports. However, in each time, only one of the combination is valid. By valid combination, we mean the combination that connects both the activated input and output ports of the process. In order to explicitly present all possible combinations between the input ports and output ports, a process is decomposed as shown in Figure 4.1.

In the original CDFD, the port list on the left side of a process is input port list, in which each input port is ordered from top to bottom, we use a number to label each port. The output ports of a process are labelled in the same way. In the decomposed graph, each node represents one port of the process, either an input port or an output port.

Each node in the decomposed graph has a name, consisting of three parts: the first character of the process's name, the identification of input or output port, and the port number. For example, the node named "P_i_1" indicates that it represents the first input port of process "Process". Different nodes are connected by two kinds of

edges, solid edges and dotted edges. The solid edges represent the data flows in the original CDFD and the dotted edge represents the mapping relationship inside the process. Like the data flows connecting two different processes in CDFD, the solid edges connect one input port node and one output port node that belong to different processes. Contrasts to the solid edges, the dotted edges connect the input port nodes and output port nodes that are in the same process. In practice, only one dotted edge in each process can be valid each time. It means in each time, a process receives data from the input port node and sends data from the output port node that are connected by the valid dotted edge.

To decompose an entire CDFD, every process in the CDFD should be decomposed in the same way as shown in Figure 4.1. Let us use the CDFD of simplified ATM, as shown in Figure 2.3, as a small example to illustrate how to decompose a CDFD. For the sake of duplication, we merely focus on the decomposition of process "Receive_Command". This process has two input ports and one output port. Each of the two input ports receives one data flow from outside, and the output port sends out a data flow to process "Check_Password". Therefore, there are three nodes in the decomposed graph representing these three ports, and three solid edges corresponding to the data flows. Two of the solid edges represent the data flows received from outside. The other solid edge represents the data flow which is sent out, it connects the output port node and the node denoting the input port of "Check_Password" . In addition to the nodes and solid edges, there are two dotted edges in graph explicitly present the relation inside the process "Receive_Command".

The remaining processes in the CDFD will be decomposed in the same way. Figure 4.2 shows the decomposed graph, where the solid edges are utilized to express the valid dotted edges. The system scenario expressed in the context of Figure 4.2 is $\{withdraw\_comm\}[Receive\_Command,\ Check\_Password,\ Withdraw]\{cash\}$. In our approach, we use notation "P_{ab}" in system scenarios to represent a process with specific input and output ports that involved in the system scenario. The notation consists of two parts: "P" represents the name of the process, and "a" and "b" are both integers that denote the number of the input and output port involved, respectively. When notation "$Withdraw_{11}$" appears in a system scenario, it means that the dotted edge connecting node "W_i_1" and "W_o_1" is valid.

The activity of deriving all possible scenarios in CDFD can be completely converted into finding all possible paths in the corresponding graph. For each process, one and only one dotted edge must be valid each time. It

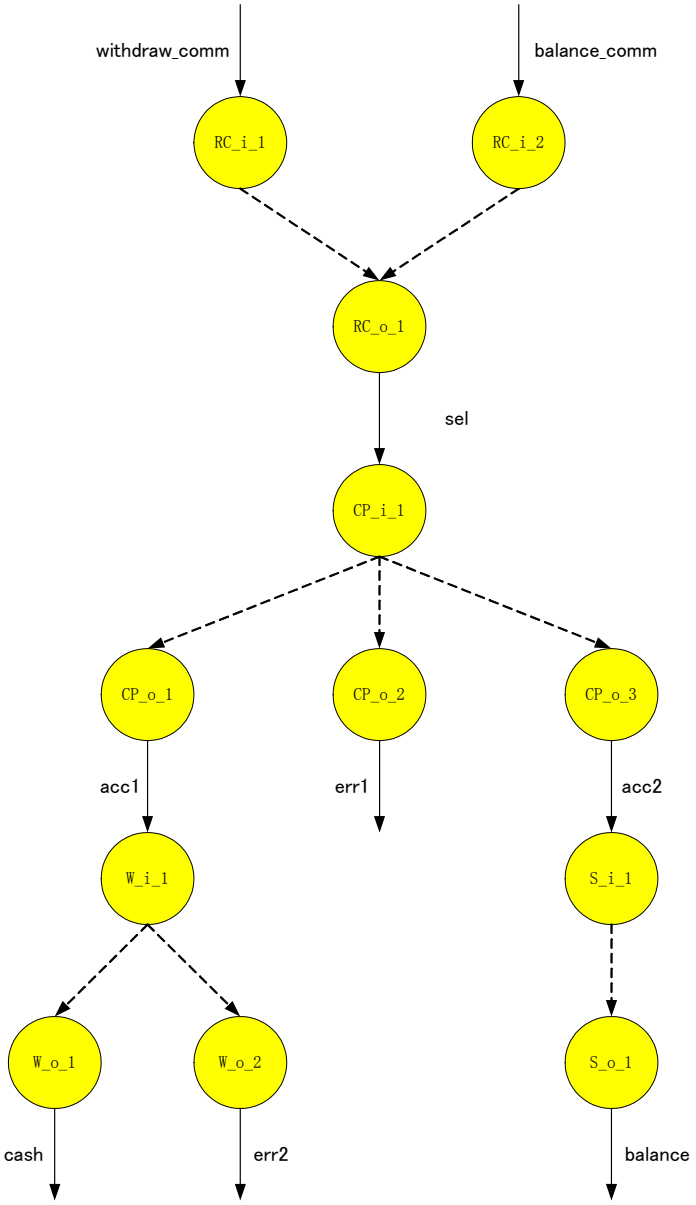


Figure 4.2: The decomposition of the CDFD of the simplified ATM

presents a specific combination of the input port and the output port. All possible system scenarios can be derived by making each dotted edge in a process valid in turn. An algorithm has been developed to automatically find all possible system scenarios. It is introduced in the next section.

4.2 Extracting System Scenarios

As mentioned previously, a system functional scenario of a CDFD is a data flow path in the CDFD. Therefore, the procedure of deriving all system scenarios can be realized by finding all possible data flow paths in a CDFD. It is equivalent to find all possible paths in the decomposed graph of the CDFD. In this section, we introduce an algorithm to derive all possible system scenarios from the decomposed graphs of CDFDs. Note that the proposed algorithm may not be effective and efficient enough to deal with the CDFDs with complicated structures, but the algorithm itself does not affect the effectiveness of our animation method. We propose this algorithm for two reasons. The first is that we want to use this algorithm to explain the basic idea for deriving all possible paths from the decomposed graph. Although we do not find any literature about finding all paths from a graph, we find this topic has been widely disscussed on many technical forums [88] [89]. However, these discussions focus on deriving paths from normal directed graphs, and cannot be directly used in deriving paths in the graph decomposed from CDFD. This is because the normal directed graphs contains only one kind of edges, but the graph decomposed from CDFD contains two kinds of edges and these two kinds of edges should be treated differently in the derivation process. The algorithm introduced in this section illustrates how these two kinds of edges should be treated and the principle underlying the algorithm can be adopted to modify the existing algorithms to effectively derive system scenarios from the CDFDs with more complicated structures. The second reason we propose the algorithm is that we want to implement a supporting tool to support our inspection method in practice. The proposed algorithm can handle most of the cases we meet in the pratice.

The structures of CDFDs that can be handled by the proposed algorithm include sequence, parallel, and simple loop. We explain the algorithm under each structure respectively.

4.2.1 Sequence Structure

Sequence structure is the simplest basic structure. Under a sequence structure, every output port of a process in a CDFD sends output data flow to only one input port of another process. In the decomposed graph, every output port node of a process connects to only one input port node that belongs to other process. It corresponds to that every output port in the original CDFD sends out data flow to only one input port or outside. The CDFD of the simplified ATM system in Figure 2.3 and its decomposed graph in Figure 4.2 have this structure. Deriving system scenarios from this structure is a straightforward process. Simply traversing the nodes connected by the solid edges and valid dotted edges in a decomposed graph can find one possible scenario. To find all possible scenarios, all possible combinations of input port nodes and output port nodes of the same process must be traversed. Therefore the valid dotted edge between one specific input port node and one specific output port node of the same process must be updated after one scenario is found. The following pseudo code describes the data structure to maintain the dotted edges of a process and the algorithm to update the valid dotted edge.

Algorithm 1 ***class*** *InputPortNode{*

```
    String processName;
    int id;
    OutputPortNode[] nexts;
    OutputPortNode nextNode;
}
void Update(Stack s){
    while (s is not empty){
        if (s.topNode is type of InputPortNode){
            int index = nexts.indexOf(nextNode)
            if (index == nexts.length - 1){
                nextNode = nexts[0];
                s.pop();
            } else {
```

```
                nextNode = nexts[index + 1];
                return;
            }
        } else {
            s.pop();
        }
    }
}
```

To explain the *Update* method, we first introduce the "*InputPortNode*" class declaration. This class represents the input port node in the decomposed graph of the CDFD. In our algorithm, we use input port node to maintain the possible combinations between the input port nodes and output port nodes in a process. The array *nexts* in the class declaration collects the output port nodes belonging to the same process as the input port node, and the output port nodes are stored in order in the array. The variable *nextNode* refers to the output port node that is connected with the input port node by valid dotted edge. In default, the valid dotted edge connects the input port node and the first output port node. In our algorithm, we use another class structure named "*OutputPortNode*" to represent the output port node. Since it has similar structure as "*InputPortNode*", we do not present it here.

Method *Update* is executed to replace the current valid dotted edge with one invalid dotted edge each time after one system scenario is derived. To avoid missing any possibility, the update process should be carried out in order. To this end, we use a stack to record the part of a system scenario that have been found. The input port node and output port node of the first process involved in the system scenario are first pushed into the stack. Therefore, the input port node and output port node of the process at the end of the system scenario are pushed on the top of the stack. In *Update* method, the algorithm first checks the type of the item on the top of the stack. If the item is an output port node, it is popped out directly. This is because the output port node does not maintain dotted edges. However, if the item is an input port node, the algorithm will decide whether the valid dotted edge connected to this input port node should be updated, or this input port node should be popped out. To make the judgement, the algorithm needs to check whether the *nextNode* of this input port node refers to the last item in the array *nexts*. If it does, that means the current valid dotted edge has presented the last possible combination between the

input port node and the output port node in the same process. Only updating the valid dotted edge of this input port node cannot help to find a new system scenario. The input port node will be popped out from the stack. On the contrary, if the *nextNode* does not refer to the last item in the array *nexts*, the algorithm updates the *nextNode* referring to the next item in the array *nexts*. The update process will not terminate until the stack is empty.

Figure 4.3 shows the example of derivation of system scenarios from the CDFD with sequence structure. The CDFD in this example contains two processes *P1* and *P2*. Process *P1* has one input port and one output port, and process *P2* has one input port and two output ports. This CDFD is first decomposed into a graph containing five nodes. Then, our automatic system scenario derivation algorithm is used to find all possible paths in the decomposed graph. The states of the stack shown in Figure 4.3 demonstrate the derivation process. The stack is used to record system scenario, and its state is changed ten times to find the system scenarios.

The fourth state of the stack shows that the first possible system scenario is found. Then the *Update* method is executed to help to find another system scenario. The node *P2_ o_ 1* is popped up in the fifth state of the stack because it is an output port node. In state six, *Update* method sets the dotted edge between *P2_ i_ 1* and *P2_ o_ 2* to be valid, so that the second possible system scenario is found.

4.2.2 Parallel Structure

The parallel structure is a structure to facilitate the description of different computations. It allows the output port in a process to send output data flows to more than one input port that belongs to different processes in a CDFD. In the corresponding decomposed graph, a output port node can connect to more than one input port node via solid edges. The CDFD in Figure 4.4 is a CDFD with parallel structure. Note that the process *P1* sends out two data flows to process *P2* and *P3*, respectively. The process *P4* can be executed as long as it receives the data flow sent from the second output port of *P2* and the data flow sent from the first output port of *P3*.

The process of deriving system scenarios from the CDFD with parallel structure is similar to deriving scenarios from sequence structure. However, there is only one additional condition that needs to be checked. This condition is that each input port node in the decomposed graph can be traversed if and only if all of the previous output port nodes that connect with it by solid edges have been traversed.

Checking the condition is necessary to ensure that every input port node can be traversed. This is because the

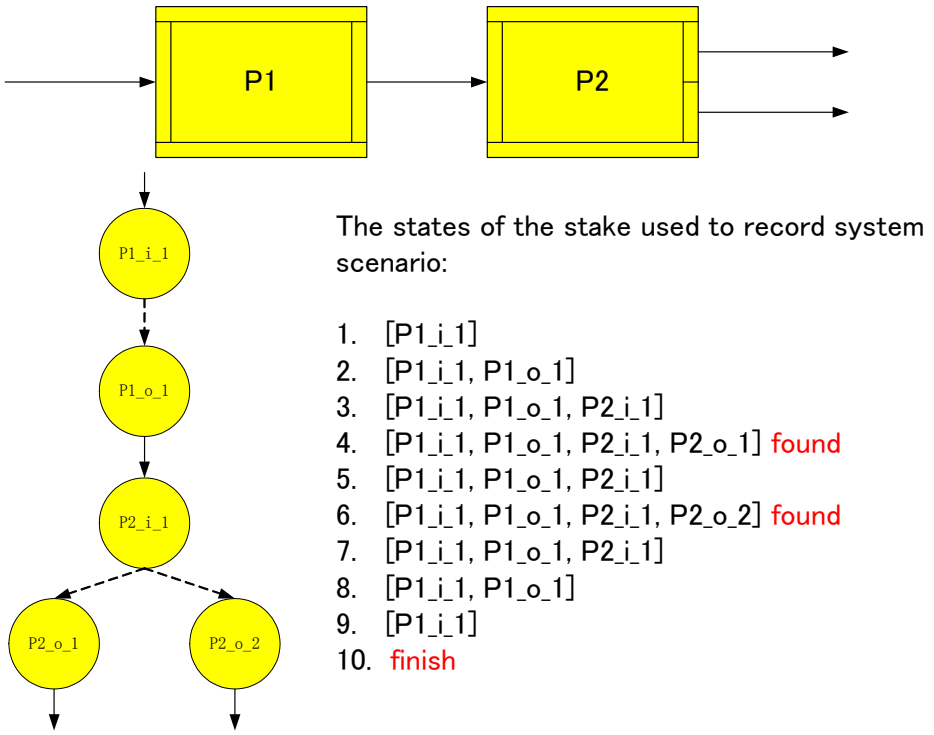
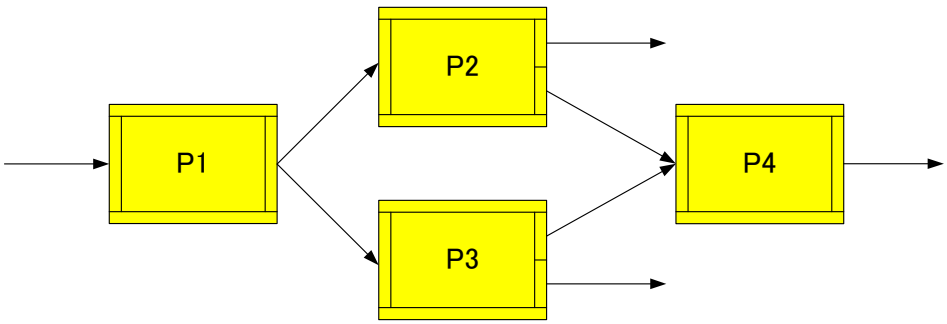


Figure 4.3: Derivation of system scenarios from the CDFD with sequence structure

semantics of a process requires all its input data flows must be available before the process can be executed. That means an input port node cannot be traversed semantically if one of its input data flow is unavailable. Checking whether all necessary input data flows are available is equivalent to checking whether all the data flows have been generated. In the decomposed graph of the CDFD, checking whether a data flow has been generated can be realized by checking whether the output port node connected by the data flow has been traversed. If the output port node has been traversed, the data flow that is sent out from this output port is considered having been generated. For example, the only input port node of process *P4* in Figure 4.4, *P4_i_1*, is connected to output port nodes *P2_o_2* and *P3_o_1* via solid edges (data flows). It can be traversed if and only if both *P2_o_2* and *P3_o_1* are traversed. If one of these two output port nodes has not been traversed, the input port node *P4_i_1* cannot be traversed since not all its input data flows are available.

In order to address the issue mentioned above, we use a modified depth-first algorithm for deriving system scenarios. In this modified algorithm, before an input port node can be traversed, whether the output port nodes that connect with it by solid edges have been traversed will be checked. If all of the output port nodes have been



The states of the stake used to record system scenario:

1. [P1_i_1]
2. [P1_i_1, P1_o_1]
3. [P1_i_1, P1_o_1, P2_i_1]
4. [P1_i_1, P1_o_1, P2_i_1, P2_o_1]
5. [P1_i_1, P1_o_1, P2_i_1, P2_o_1, P3_i_1]
6. [P1_i_1, P1_o_1, P2_i_1, P2_o_1, P3_i_1, P3_o_1]
7. [P1_i_1, P1_o_1, P2_i_1, P2_o_1, P3_i_1]
8. [P1_i_1, P1_o_1, P2_i_1, P2_o_1, P3_i_1, P3_o_2] found
9. [P1_i_1, P1_o_1, P2_i_1, P2_o_1, P3_i_1]
10. [P1_i_1, P1_o_1, P2_i_1, P2_o_1]
11. [P1_i_1, P1_o_1, P2_i_1]
12. [P1_i_1, P1_o_1, P2_i_1, P2_o_2]
13. [P1_i_1, P1_o_1, P2_i_1, P2_o_2, P3_i_1]
14. [P1_i_1, P1_o_1, P2_i_1, P2_o_2, P3_i_1, P3_o_1]
15. [P1_i_1, P1_o_1, P2_i_1, P2_o_2, P3_i_1, P3_o_1, P4_i_1]
16. [P1_i_1, P1_o_1, P2_i_1, P2_o_2, P3_i_1, P3_o_1, P4_i_1, P4_o_1] found
17. [P1_i_1, P1_o_1, P2_i_1, P2_o_2, P3_i_1, P3_o_1, P4_i_1]
18. [P1_i_1, P1_o_1, P2_i_1, P2_o_2, P3_i_1, P3_o_1]
19. [P1_i_1, P1_o_1, P2_i_1, P2_o_2, P3_i_1]
20. [P1_i_1, P1_o_1, P2_i_1, P2_o_2, P3_i_1, P3_o_2]
21. [P1_i_1, P1_o_1, P2_i_1, P2_o_2, P3_i_1]
22. [P1_i_1, P1_o_1, P2_i_1, P2_o_2]
23. [P1_i_1, P1_o_1, P2_i_1]
24. [P1_i_1, P1_o_1]
25. [P1_i_1]
26. finish

Figure 4.4: Derivation of system scenarios from the CDFD with parallel structure

traversed, the input port node will be traversed and pushed into the stack as a part of current system scenario. Otherwise the algorithm traverses the first node in the buffer or terminates the derivation process. The reason why our algorithm underlies the depth-first algorithm is that the depth-first algorithm can traverses the nodes in their execution order.

Due to the depth-first algorithm is a well-known algorithm and the modification is not very huge, we merely give an example in Figure 4.4 rather than describe the algorithm in detail. The *Update* method used in sequence structure can also be used in parallel structure. In this example, the state of the stack is changed 26 times to derive two system functional scenarios. In the sixth state of the stack, node *P3_o_1* is traversed. However, node *P4_i_1* cannot be traversed since node *P2_o_2* is not traversed and there is no node in the buffer. Then the *Update* method is executed to update the valid dotted edge in process *P3*, and the first system scenario is found in the eighth state of the stack.

4.2.3 Loop Structure

Loop structure is the most complicated one in the three structures. It may contain sequence or parallel structures. Before discussing the derivation algorithm, we first define what the loop structure is. The loop structure in the decomposed graph of a CDFD is defined as follows:

- The beginning of a loop is the output port node that sends out a data flow to an input port node that has already existed in one system scenario.
- The loop structure is a series of nodes, including input and output port nodes. The end of a loop structure is the input port node that connects to the beginning output port node by solid edge in the system scenario.

For example, the CDFD shown in Figure 4.5 contians a loop. In the CDFD, the second output port of process "*P3*" sends out a data flow to the second input port of process "*P1*". Since the process "*P1*" is executed before the execution of process "*P3*", a loop is caused by the second output port of "*P3*". In the decomposed graph, the loop starts from the node *P3_o_2*, which presents the second output port of "*P3*", and the end of the loop is the node *P3_i_1* since it is connected to the beginning node *P3_o_2* by valid dotted edge when the loop is caused.

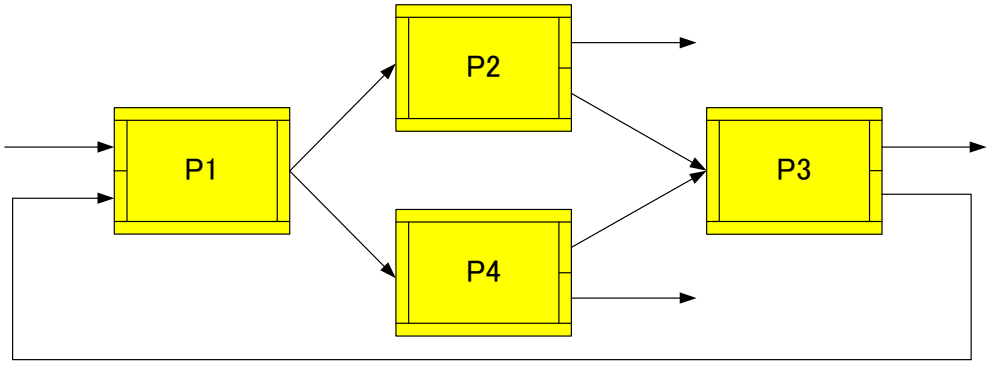


Figure 4.5: CDFD with loop structure

Since the purpose of system scenario generation is to perform animation rather than execut the formal specification, the series of nodes of a loop will only appear once in each system scenario and it is enough for the user to observe the behavior of the system in the animation.

To teminate a loop in the system scenario, the algorithm needs to ensure that the output port node that causes the loop cannot be visited again in the scenario. For example, the loop in the CDFD shown in Figure 4.5 is caused by node *P3_ o_ 2*, therefore, this node will not be visited again when the input port node *P3_ i_ 1* is visited.

The loop structure will be represented in at least one system scenario. One of such scenarios in the CDFD shown in Figure 4.5 is $[P1_{11}, P2_{12}, P4_{11}, P3_{12}, P1_{21}, P2_{12}, P4_{11}, P3_{11}]$. The starting node of the loop structure is *P3_ o_ 2*, and the end node is *P3_ i_ 1*. Since the loop can only be presented once in the system scenario, the output port node *P3_ o_ 2* will not be traversed again.

Because each node might be traversed more than once in the loop structure, the node class structure we discussed in the previous section is not suitable. It needs to be improved to deal with the loop structure. In addition to the modification to node class structure, the *Update* method also needs to be modified to handle the loop structure. Since the output port node causing a loop structure cannot be traversed again after the loop is terminated, the afterward update process of invalid dotted edges should avoid the end node. The original *Update* method is not adapted for handling this issue.

Algorithm 2 ***class*** *InputPortNode{*

String *processName;*

```
    int id;
    OutputPortNode[] nexts;
    OutputPortNode[] nextNodeEachTime;

    //the following variables is used to handle the loop structure
    bool inLoop;
    OutputPortNode causedNode;
    int startLocation;
    OutputPortNode[] nextsInLoop;
}
void Update (Stack s) {
    while (s is not empty) {
        if(s.topNode is type of InputPortNode) {
            int length = nextNodeEachTime.length;
            if(s.topNode.inLoop) {
                if(s.topNode.startLocation == length) {
                    s.topNode.inLoop = false;
                    int index = nexts.indexOf(nextNodeEachTime[length - 1]);
                    if(index == nexts.length - 1){
                        nextNodeEachTime.removeIndexOf(length - 1);
                        s.pop();
                        continue;
                    } else {
                        nextNodeEachTime[length - 1] = nexts[index + 1];
                        return;
                    }
                }
```

```
            } else {
                int index = nextsInLoop.indexOf(nextNodeEachTime [length - 1]);
                if(index == nextsInLoop.length - 1){
                    nextNodeEachTime.removeIndexOf(length - 1);
                    s.pop();
                    continue;
                } else {
                    nextNodeEachTime[length - 1] = nextsInLoop[index + 1];
                    return;
                }
            } else {
                int index = nexts.indexOf(nextNodeEachTime[length - 1]);
                if(index == nexts.length - 1){
                    nextNodeEachTime.removeIndexOf(length - 1);
                    s.pop();
                    continue;
                } else {
                    nextNodeEachTime[length - 1] = nexts[index + 1];
                    return;
                }
            }
        }
    } else {
        s.pop();
        continue;
    }
}
```

}

Since in a loop structure, an input port node may appear more than one time in a system scenario, an array *nextNodeEachTime* is used to replace *nextNode* in the improved class structure to maintain the valid dotted edges each time the input port node appears. Moreover, four variables are added to the improved class declaration for recording the information of loop structure. The first variable is boolean typed *isLoop*, which is used to record whether a loop is started. It is necessary since after a loop is started, we need to avoid the start node of the loop being traversed again. The second variable *causedNode* is used to record the start node of the loop structure. The third is an integer typed variable *startLocation*, which is used to record how many times an input port node has appeared before one of its connected output port node causes a loop. The last variable named *nextsInLoop* is an array that stores all of the output port nodes belonging to the same process in order except the node that causes the loop.

The refined *Update* method is used to update the valid dotted edges and avoid the start node of a loop structure. The essential idea of this refined method is the same as the original one, and it can also deal with the sequence and parallel structures. The modification in the refined method is adding a judgment before updating an invalid dotted edge. If *inLoop* is *true*, that means the current appearance of the input port node is inside a loop structure, and the new valid dotted edge should avoide the output port node that causes the loop. Therefore, the dotted edge that would be set valid should connect with an output port node in the array *nextsInLoop*. If the current output port node connected by the valid dotted edge refers to the last node in *nextsInLoop*, that means all the possible combinations have been explored. The current input port node will be popped out from the top of the stake. The rest of the method is almost the same as the original one.

We use the CDFD shown in Figure 4.5 as an example to illustrate the derivation process under loop structure. Since the entire derivation process is too long to be presented here, we merely explain part of the entire process for illustration. Assuming the following system scenario has already been derived:

$$[P1_{11},\ P2_{12},\ P4_{11},\ P3_{11}]$$

The last item $P3_{11}$ in the system scenario indicates that the last process in the system scenario is process *P3*, and it receives input data flow from its first input port and sends out data flow from its first output port.

Correspondingly, in the first two nodes on the top of the stake in the algorithm is *P3_ i_ 1* and *P3_ o_ 1*. By executing the *Update* method, the output port node *P3_ o_ 1* is popped out first, then the valid dotted edge connected to the input port node *P3_ i_ 1* is updated to connect the output port node *P3_ o_ 2*. Since the node *P3_ o_ 2* sends out a data flow to input port node *P1_ i_ 2*, which has already been traversed in the current system scenario, a loop is caused.

Therefore, the variables of node *P3_ i_ 1* used to record loop structure information are initialized respectively. The variable *inLoop* is assigned value *true* and the variable *causedNode* refers to output port node *P3_ o_ 2* that causes the loop. The value of *startLocation* is *1* because the input port node *P3_ i_ 1* appears only one time when the loop is started. The last variable *nextsInLoop* contains all the output port nodes of process *P3* except the output port node *P3_ o_ 2*.

By traversing the nodes following the solid edges and updated valid dotted edges, the following system scenario can be derived:

$$[P1_{11},\ P2_{12},\ P4_{11},\ P3_{12},\ P1_{21},\ P2_{11},\ P4_{12}]$$

Then, the *Update* method is executed again and the following system scenario is derived:

$$[P1_{11},\ P2_{12},\ P4_{11},\ P3_{12},\ P1_{21},\ P2_{12},\ P4_{11},\ P3_{11}]$$

To derive another possible system scenario, the *Update* method first pops out *P3_ o_ 1*, which is the node on the top of the stack. The input port node *P3_ i_ 1* is popped out next rather than updated. This is because the *isLoop* variable of node *P3_ i_ 1* is true. That means the node *P3_ i_ 1* is in a loop and the valid dotted edge should be updated to connect to the output port node in array *nextsInLoop*. Since the node *P3_ o_ 1* is the only item in *nextsInLoop*, the node *P3_ i_ 1* is popped out.

The *Update* method is executed continuously until all the nodes are popped out from the stack and no new possible system scenario is found.

4.2.4 Limitation of the Algorithm

In the algorithm introduced in the previous section, we assume that no nested loop structure existing in the CDFD. We make this assumption since in our experience almost all of the CDFDs do not include the nested loop structure.

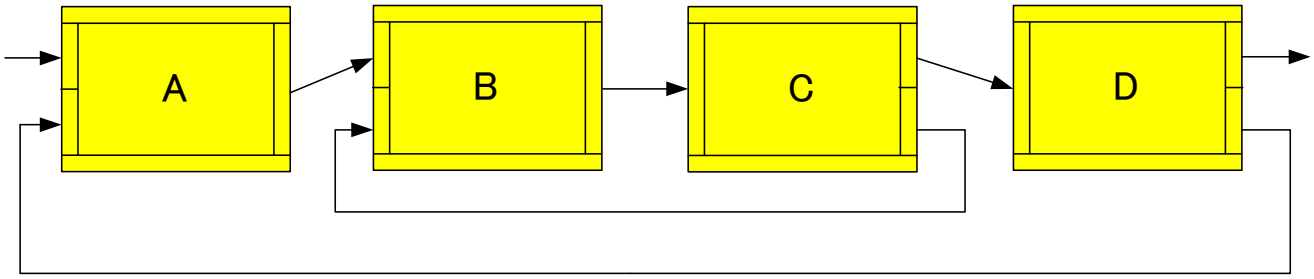


Figure 4.6: CDFD with nested loop

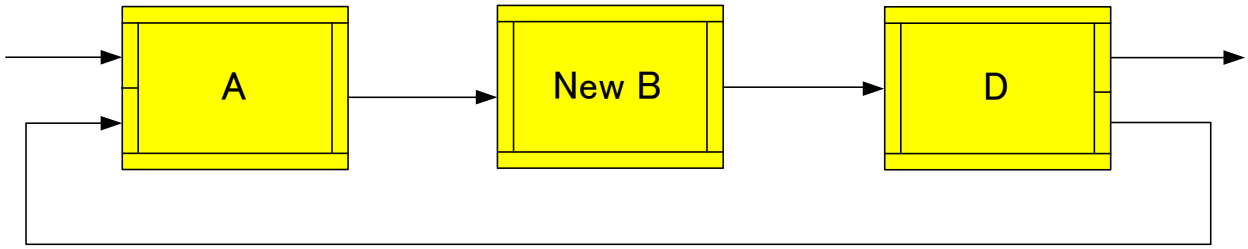


Figure 4.7: A new CDFD by combining process "B" and "C"

Although it is correct in syntax, in practice it will make the whole CDFD very complex and few designers will use it. The CDFD with nested loop structure and its associated module specification can confuse not only the users but also the developers.

In SOFL practice, we suggest using two levels of CDFDs to present the functionality of a nested loop structure. For example, Figure 4.6 shows a CDFD with nested loop. By combining the process "*B*" and "*C*" into a new process "*New B*", the CDFD shown in Figure 4.7 can be constructed. To present the functionality of the CDFD shown in Figure 4.6, the pocess "*New B*" should be decomposed into a new CDFD, as shown in Figure 4.8. The two CDFDs in Figure 4.8 describes the same functionality as the CDFD in Figure 4.6 but have clearer structures. To facilitate the user to decompose process, the supporting tool developed for SOFL approach provides a function to allow the user decompose a process from the CDFD directly (see Chapter 7 for detail).

Except for the nested loop, the proposed algorithm is able to derive system scenarios from simple loop structure and sequential loop structure. Figure 4.9 shows a CDFD with sequential loop structure. Two loops "Loop A" and "Loop B" exist in the CDFD and they are executed sequentially. By using the proposed algorithm, the following

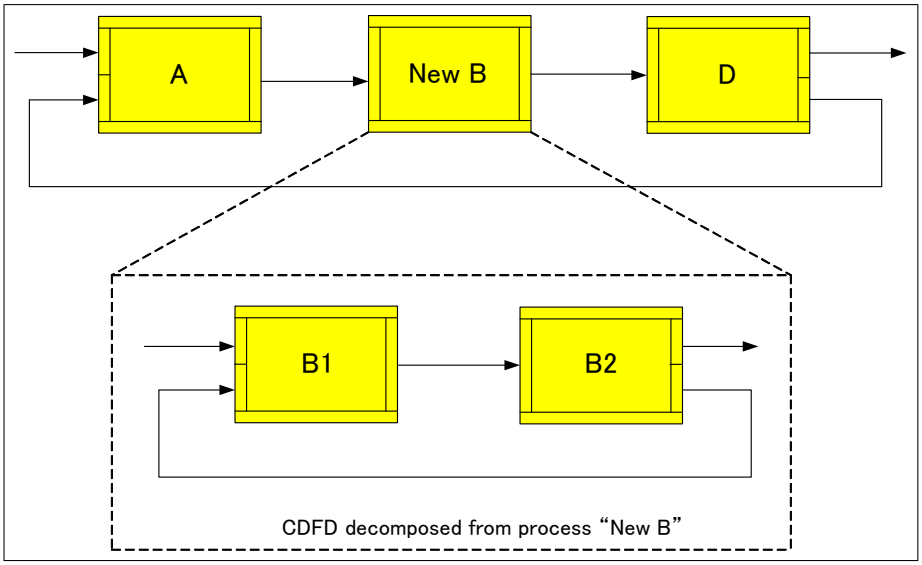


Figure 4.8: Decomposing the process "New B"

four system scenarios can be derived:

1. $[A_{11}, B_{11}, C_{11}, D_{11}]$
2. $[A_{11}, B_{11}, C_{11}, D_{12}, C_{21}, D_{11}]$
3. $[A_{11}, B_{12}, A_{21}, B_{11}, C_{11}, D_{11}]$
4. $[A_{11}, B_{12}, A_{21}, B_{11}, C_{11}, D_{12}, C_{21}, D_{11}]$

The proposed algorithm has been implemented as part of our supporting tool introduced in Chapter 7. Based on the testing results, we believe the algorithm can be used to derive all possible system scenarios for CDFDs that do no contain nested loop. The results of generating system scenarios from the CDFDs shown in Figure 4.5 and 4.9 will be demonstrated in Chapter 7.3.

4.2.5 Combinatorial Explorsion

The essential idea of the proposed algorithm is to explore every possible combination between input ports and output ports. With the increasing of the complexity of the CDFD, the number of the combinations between different ports will increase repidly. The combinatorial explorsion can significantly affect the efficiency of the

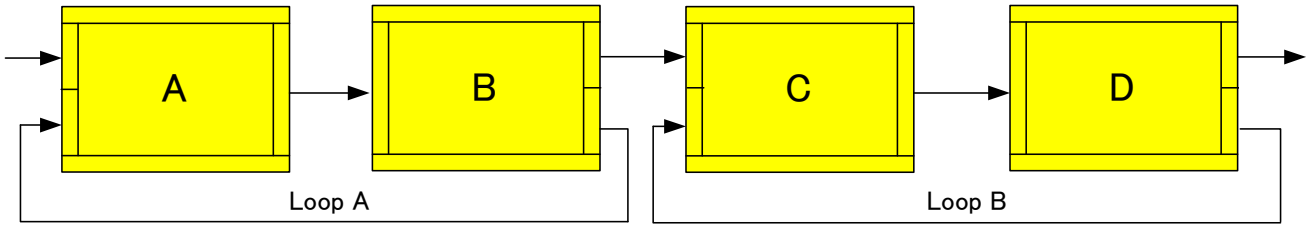


Figure 4.9: Sequential Loops

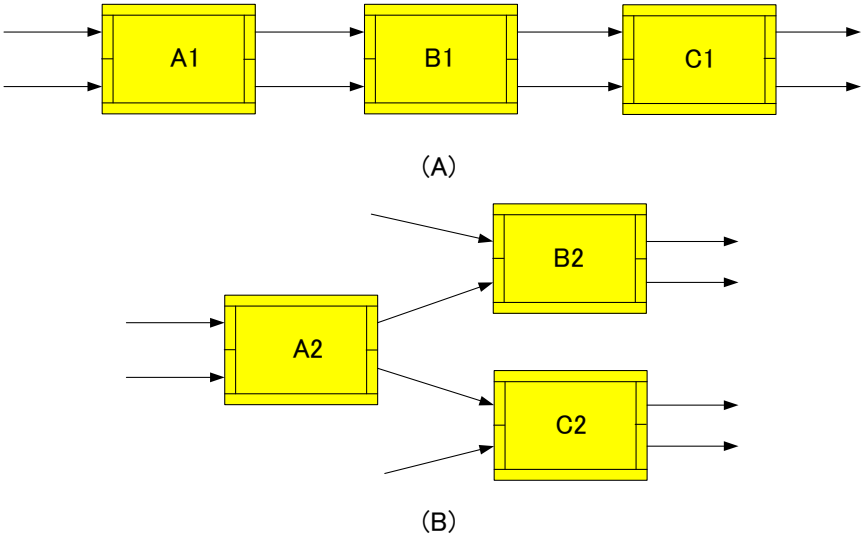


Figure 4.10: Two CDFDs with the same number of processes

proposed algorithm. Unfortunately, there is not general formula to estimate how many system scenarios can be produced from a CDFD merely based on the number of processes and ports. Even for two CDFDs with the same number of processes and input and output ports, the number of system scenarios of these two CDFDs may be different due to their different structures.

For example, Figure 4.10 shows two CDFDs that both have three processes and each process has two input ports and two output ports. However, these two CDFDs contain different numbers of system scenarios. Figure 4.10 (A) shows a CDFD with three processes that are sequentially connected. In this CDFD, 16 system scenarios can be produced. Figure 4.10 (B) shows a CDFD with parallel structure, which also has three processes. The number of system scenarios that can be produced form CDFD (B) is 12, which is different from CDFD (A).

Although the complexity of a control flow diagram has been estimated in some literatures [90] [91], there is no

existing theory for estimating the complexity of a data flow diagram. Instead, we use testing rather than a general formula to estimate the number of system scenarios in a CDFD, the test results will be reported in Chapter 7.3. Meanwhile, we belive the maxmum number of system scenarios in a CDFD can be calculated by follows.

Assume a CDFD contains n processes, and each process P_i ($i \in \{1, 2, ..., n\}$) has a_i ($a_i \in [1, +\infty)$) input ports and b_i ($b_i \in [1, +\infty)$) output ports, then, the number of system scenarios contained in the CDFD will not be more than:

$$\prod_{i=1}^{n} a_i \times b_i$$

Based on our experience, the real number of system scenarios of a CDFD is far less than the number calculated using the above formula. In Chapter 7.3, we use a CDFD containing 30 processes to evaluate the combinatorial explorsion problem of the proposed algorithm and the result indicates that a huge number of possible system scenarios will be generated from a CDFD with complex structure.

Since the system scenario generation algorithm is not the most concern of our research, we did not formally prove and evaluate its effectiveness and efficiency in this dissertation. The aim of proposing the algorithm is to explain the basic idea of finding system scenarios in CDFDs and handle the most common cases we met in practice. In the future, we plan to investigate the existing algorithms to find a more effective and efficient algorithm for generating possible system scenarios.

4.3 Animating Specifications

As introduced previously, the formal specification can be reorganized into a collection of mutually exclusive and collectively exhaustive system functional scenarios. Each system scenario represents an independent behavior of the software system described in the formal specification. To dynamically present the potential behaviors of the entire formal specification, we suggest that every possible system scenario defined in the specification should be animated. In this section, we introduce the procedure to animate a formal specification.

4.3.1 Animation Process

For a given formal specification, the following steps supply a procedure for systematically performing the animation.

Step 1: Derive all possible *system scenarios* from the formal specification.

An algorithm has been introduced in the previous sections to derive system scenarios from the topology of a CDFD. This algorithm can be used in Step 1, and a supporting tool has been implemented based on the algorithm to automatically derive system scenarios.

Before further discussion, the concept of *operation functional scenario* must be introduced first since it is used in the second step. An operation scenario is a conjunction of *pre-condition*, *guard condition*, and *defining condition* that is extracted from the pre- and post-conditions of a process defined in the module; it defines an independent behavior of the process in terms of input and output relation. Let $\mathrm{P}(\mathrm{P}_{iv}, \mathrm{P}_{ov})[\mathrm{P}_{pre}, \mathrm{P}_{post}]$ denote the process specification of a process P, where P_{iv} is the set of all input variables whose values are not changed by the process, P_{ov} is the set of all output variables whose values are produced or updated by the process, and P_{pre} and P_{post} are the pre and post-conditions of the process P, respectively. The following is the definition of *operation functional scenario.*

Definition 2 *Let* $P_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \ldots \vee (C_n \wedge D_n)$*, where each* C_i *(*$i \in$ *{1, ..., n}) is a predicate called "guard condition" that contains no output variable in* P_{ov} *and* $forall_{i,j} \in \{1, \ldots, n\} \cdot i \neq j \Rightarrow C_i \wedge C_j = false$*; *$D_i$ *a "defining condition" that contains at least one output variable in* P_{ov} *but no guard condition. Then, the formal specification of a process can be expressed as a disjunction* $(\breve{}P_{pre} \wedge C_1 \wedge D_1) \vee (\breve{}P_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (\breve{}P_{pre} \wedge C_n \wedge D_n)$*. A conjunction* $\breve{}P_{pre} \wedge C_i \wedge D_i$ *is called an operation functional scenario, or operation scenario for short.*

Note that we use $\breve{}x$ and x to represent the initial value of state variable x before and after the process respectively. The decorated pre-condition $\breve{}P_{pre} = P_{pre}[\breve{}x/x]$ denotes the predicate resulting from substituting the initial state $\breve{}x$ for the final state x in the pre-condition P_{pre}. For example, three operation scenarios are listed in Table 4.1.

Step 2: Let $\{\mathrm{d}_i\}[\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_n]\{\mathrm{d}_o\}$ be a selected system scenario. Derive appropriate *operation scenarios* of each process P_i (i $\in$ {1, 2, ..., n}) from the specification and get a set of *operation scenarios* $\{\mathrm{OS}_1, \mathrm{OS}_2, \ldots, \mathrm{OS}_n\}$, where OS_i is the appropriate *operation scenario* of process P_i.

According to our animation strategy, each time only one system scenario should be selected to animate. For any selected system scenario, the related *operation scenarios* of each process involved should be derived from the

```
process Check_Password(id: string, sel: bool, pass: string)
                    acc1: Account | err1: string | acc2: Account
ext    rd #Account_file
pre    true
post   (exists![x: Account_file] | ((x.id = id and x.password = pass) and
       (sel = true and acc1 = x or sel = false and acc2 = x)))
       or
       not(exists![x: Account_file] | (x.id = id and x.password = pass)) and
         err1 = "Reenter your password or insert the correct card"
end_process
```

Figure 4.11: The definition of process "Check_Password"

formal specification. Since an *operation scenario* defines an independent relation between the input and output of a process, only one *operation scenario* of a process can be involved in a specific system scenario. Therefore, the second step of the entire animation process should be deriving appropriate operation scenarios.

Using the process specification of "Check_Password" as an example. This process belongs to the formal specification of the simplified ATM system in Figure 2.2. Figure 4.11 shows the definition of the process "Check_Password". According to Definition 2, three operation scenarios are defined in this process, which are listed in Table 4.1. Supposing the system scenario that is currently under animation is {*withdraw_comm*} [*Receive_Command, Check_Password, Withdraw*] {*cash*}, the context of this system scenario indicates that the process "Check_Password" sends out data to process "Withdraw". Therefore, only the first operation scenario listed in Tabel 4.1 is involved in the current system scenario.

Step 3: Collect appropriate test suites that satisfy each operation scenario OS_i derived in Step 2.

As introduced previously, our animation is done by means of "executing" a system scenario. However, the "execution" is not to actually run the formal specification on a computer. Instead, we use data to connect each process involved in the system scenario and demonstrate how the input data are produced to the output data. The collected test suites are the data used to animate the system scenario. A test suite includes the input data

Table 4.1: Three operation scenarios in the process

Number	**Test cases**
1	true and exists![x: Account_file] | ((x.id = id and x.password = pass) and sel = true and acc1 = x)
2	true and exists![x: Account_file] | ((x.id = id and x.password = pass) and sel = false and acc2 = x)
3	true and not(exists![x: Account_file] | (x.id = id and x.password = pass)) and err1 = "Reenter your password or insert the correct card"

(test case) and output data (expected result). In principle, the collected test suites should satisfy the operation scenario and represent the most concerns of the user. The details will be introduced in the next section.

Step 4: Use the test suites collected in Step 3 to execute the selected system functional scenario.

In this step, the processes in the system scenario are "executed" in turn. The execution is presented on the corresponding CDFD, and the data flows used to connect involved processes are replaced by the test suite in "execution" order. This process will be illustrated in the following section.

Step 5: Repeat Step 2 to Step 4 until all the system scenarios derived in Step 1 are animated.

Animating all possible behaviors of the system is required by our animation method. The process of animating a specific behavior is described from Step 2 to Step 4 above.

4.3.2 Test Suite Selection

Animation of a CDFD is done by animating each individual system scenario derived. Animation of a system scenario in this case is actually a dynamic "execution" of the scenario, By "execution", we mean a dynamic demonstration of what input values are used to lead to what output values. To perform such an animation, we need both input and output values.

The input and output values for an animation are known as *test case* and *expected result*, respectively, throughout this dissertation. A test case and the corresponding expected result together are called a *test suite*. Since the

purpose of animation in our approach is to serve as a reading technique to facilitate inspection of the scenario against the corresponding informal requirement, a test suite should be generated based on the informal requirements specification. Furthermore, due to the fact that the judgement on whether errors are found during the process of animation-based inspection usually needs to be made by both the designer and the user through comparing the animation result to the informal requirements, the test suite should usually be made by both the designer and the user in collaboration. The first thing to do in the animation is to substitute the test case for the corresponding input variables in the pre-condition and the expected result for the corresponding output variables in the post-condition to automatically check whether they satisfy the pre- and post-conditions of the process to be animated. Only the satisfied test suite is used for dynamic demonstration.

An alternative way to collect the test suite satisfying the pre- and post-conditions is automatically generating the test suite. According to the previous section, only one of the operation scenarios of a process is involved in the system scenario. Therefore, for a specific system scenario, the generated test suite must satisfy the related operation scenario rather than other operation scenarios in the process. The generation can be separated into the generation of test case and the generation of expected result. An automatic test case generation method have been given in [92], however, it did not indicate how to generate expected result. We extend the original method to deal with our problem. The following procedure can be used to generate test suites.

Stage 1: Eliminate Defining condition. The defining condition D_i is eliminated first since it usually does not provide the useful information for test case generation. Based on the definition of operation scenario, as the input data of a process, the test case should be generated based on the conjunction of pre-condition and guard condition $\tilde{}P_{pre} \wedge C_i$.

Stage 2: Convert to disjunctive normal form. The conjunction of pre-condition and guard condition $\tilde{}P_{pre} \wedge C_i$ is transformed into an equivalent disjunctive normal form (DNF) with form $P_1 \vee P_2 \vee ... \vee P_n$. Each P_i is a conjunction of atomic predicate expressions, say $Q_1^i \wedge Q_2^i \wedge ... \wedge Q_m^i$.

Let $Q(x_1, x_2, ..., x_w)$ be one of the atomic predicate expressions Q_1^i, Q_2^i, ..., Q_m^i mentioned previously. The variables x_1, x_2, ..., x_w is a subset of all the input variables. The values for the input variables involved in each atomic predicate expression Q can be generated using an algorithm that deals with the following three situations,

Table 4.2: Input data generation algorithm

Number	$\ominus$	Algorithms of test case generation for x_1
1	$=$	$x_1 = E$
2	$<$	$x_1 = E + \Delta x$
3	$>$	$x_1 = E - \Delta x$
4	$\leq, \geq, \neq$	similar to above

respectively. Here we are using variables of numerical types as examples for convenience.

Situation 1: If only one input variable is involved and $Q(x_1)$ has the format $x_1 \ominus E$, where $\ominus \in \{=, <, >, \leq, \geq, \neq\}$ is a relational operator and E is a constant expression, using the algorithms listed in Table 4.2 to generate test cases for variable x_1.

Situation 2: If only one input variable is involved and $Q(x_1)$ has the format $E_1 \ominus E_2$, where E_1 and E_2 are both arithmetic expressions which may involve variable x_1. This atomic predicate is first transformed to the format $x_1 \ominus E$. Then apply the algorithm in Table 4.2 to generate test cases for variable x_1.

Situation 3: If more than one input variables are involved and $Q(x_1, x_2, ..., x_w)$ has the format $E_1 \ominus E_2$, where E_1 and E_2 are both arithmetic expressions possibly involving all the variables $x_1, x_2, ..., x_w$. First randomly assigning values from appropriate types to the input variables $x_2, x_3, ..., x_w$ to transform the format into the format $E_1' \ominus E_2'$ that contains only one input variable x_1, then generate the test case for x_1 as stated in Situation 2.

Usually, more than one atomic predicate may be contained in an operation scenario and one variable may be included in more than one atomic predicate. For example, the conjunction of pre- and guard conditions of an operation scenario has the form $x < y \wedge y < z$. Two atomic predicates and three input variables are included, and variable y is in both predicates. The generated value for variable y must satisfy both predicates in the meantime.

An expected result corresponds to the test case in the same test suite. It is the output of an operation scenario that receives the test case. After having the test case, the expected result can be generated by applying the test case to the operation scenario. The following procedure is used to generate expected result.

Stage 3: Replace input variables. Replace the input variables in the operation scenario with the generated test case, and get a new predicate $\tilde{}P'_{pre} \wedge C'_i \wedge D'_i$, which merely contains output variables.

Stage 4: Convert to disjunctive normal form. Convert the conjunction $\tilde{}P'_{pre} \wedge C'_i \wedge D'_i$ into an equivalent disjunctive normal form (DNF) with form $P'_1 \vee P'_2 \vee ... \vee P'_n$. Each P'_i is a conjunction of atomic predicate expressions, say $Q^{i\prime}_1 \wedge Q^{i\prime}_2 \wedge ... \wedge Q^{i\prime}_m$.

The conjunction $Q^{i\prime}_1 \wedge Q^{i\prime}_2 \wedge ... \wedge Q^{i\prime}_m$ has the same form as the conjunction $Q^i_1 \wedge Q^i_2 \wedge ... \wedge Q^i_m$ explained previously. The only difference is that the former contains only output variables and the latter includes only input variables. The three situations and the algorithm listed in Table 4.2 can also be used for generating expected result.

Note that a process, except the first one in a system scenario, may receive input data from two origins. One origin is its previous process, and the other is outside. Supposing the system scenario {*withdraw_comm*} [*Receive_Command, Check_Password, Withdraw*] {*cash*} is selected for animation. The process "*Withdraw*" is the last process in this system scenario and its previous process is "*Check_Password*". Figure 4.12 shows that the process "*Withdraw*" receives two input variables "*amount*" and "*acc1*". Considering the definition of process "*Check_Password*" shown in Figure 4.11, the input variable "*acc1*" of process "*Withdraw*" is also the output variable of process "*Check_Password*". When generating test case for "*Withdraw*", only the value of variable "*amount*" is generated. The value of variable "*acc1*" is received from "*Check_Password*", and the value is actually the expected result of "*Check_Password*".

According to the context of the selected system scenario, the operation system of process "*Withdraw*" involved in the system scenario is "*true and amount <= ~acc1.balance and acc1.balance = ~acc1.balance - amount and cash = amount*". We use the pre-condition and guard condition to generate test case for this operation scenario as shown in the following:

$$true\ and\ amount <= \tilde{}acc1.balance$$

In this predicate expression, the value of variable "*acc1*" is predetermined and received from process "*Check_Password*". The type of "*acc1*" is "*Account*" that is a compound type and contains four fields, "*id*", "*name*", "*password*", *and* "*balance*", as shown in Figure 2.2. Assuming the value of "*acc1*" received from "*Check_Password*" is (0001,

```
process Withdraw(amount: nat0, acc1: Account) err2: string | cash: nat0
ext    wr #Account_file
pre    true
post   amount <= ~x.balance and x.balance = ~x.balance - amount and
           cash = amount
       or
       amount > ~x.balance and err2 = "You do NOT have enough balance!"
end_process
```

Figure 4.12: The definition of process "Withdraw"

"*Jack*J, 1111, 15000), each item in this 4-tuple corresponding to the fields in type "*Account*" respectively. By replacing the variable acc1 with its value, the above predicate expression is transformed into a new expression with only one variable.

$$true\ and\ amount <= 15000$$

According to the algorithm listed in Table 4.2, the value of "*amount*" can be automatically generated by executing the expression $15000 - \Delta x$, where Δx is a positive random number. If a positive value 332 is randomly assigned to Δx, the value of "*amount*" should be 14668. Therefore the test case for the operation scenario is {(0001, "*Jack*J, 1111, 15000), 14668}.

By replacing the input variables in the operation scenario with the test case, the original operation scenario becomes a predicate with only output variables:

$$true\ and\ 14668 <= 15000\ and\ acc1.balance = 15000 - 332\ and\ cash = 14668$$

Two output variables contained in above predicate are "*acc1.balance*" and "*cash*". The "*acc1.balance*" presents the balance of the account "*acc1*" after the withdraw is executed, and it is equal to the original balance in the bank account minus the amount that has been withdrawn. The "*cash*" is the number that is displayed to the end user to inform him the success of withdraw, and this number should be equal to the amount that has been withdrawn.

By applying the algorithm listed in Table 4.2, we can get the expected result {(0001, "*Jack*], 1111, 332), 14668}.

Although the above procedure provides a possibility for full automation in test suite generation, the effect of using the automatically generated test suite may not be satisfactory in practice because the intention of the automatically generated test case and expected result may be hard to understand by the user and/or the designer. For internal consistency checking of a system scenario, automatically generated test suite may be cost-effective because most of the internal consistency properties can be defined as a predicate expression in advance and the meaning of their evaluation is precisely defined.

A test suite should be generated to fulfill at least the following targets in order to help validation:

1. *Demonstrate a normal use case.*

2. *Demonstrate an exceptional use case when some condition of interest fails to meet.*

3. *Demonstrate extreme cases, such as boundary conditions, data items in a data structure being too many or too few.*

For example, for the animation of the first system scenario of the simplified ATM, that is, {*withdraw_comm*} [*Receive_Command, Check_Password, Withdraw*]{*cash*}, we generate the test suites in Table 4.3 manually to demonstrate a normal use case.

4.3.3 Execution of System Functional Scenarios

Execution of a system scenario is to dynamically display the situation where the selected test case as input leads to the selected expected result for each process involved in the scenario. Such an execution must be carried out sequentially. In the case of a data flow being used as both an output of a process and an input of the next adjacent process, the value produced for it must be kept consistent in use. Figure 4.13 shows a process of animating the first system scenario derived from the ATM CDFD. Figure 4.13 (A) shows the execution of the starting process "*Receive_Command*"; (B) shows the execution of the second process "*Check_Password*"; and (C) shows the execution of the terminating process "*Withdraw*". These three diagram together depicts the process of the animation of the system scenario. The test cases and the expected results used in this animation are taken from Table 4.3.

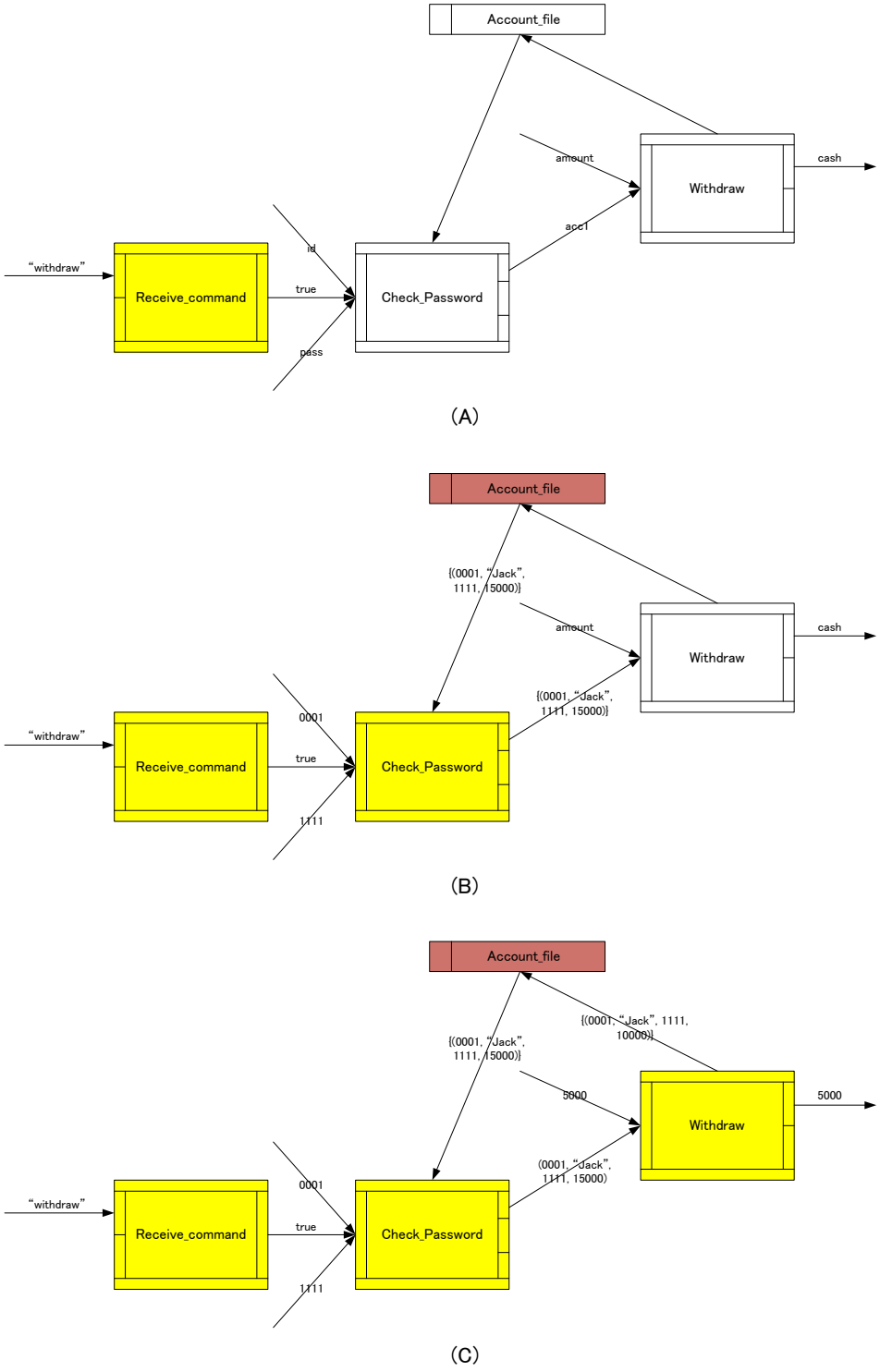


Figure 4.13: The animation process of a system functional scenario

Table 4.3: Test suites for a normal function

Input variables	**Test cases**	**Expected variables**	**Expected results**
withdraw_comm	"withdraw"	*sel*	true
id	0001		
sel	true		
pass	1111	*acc1*	(0001, "Jack" 1111, 15000)
Account_file	{(0001, "Jack" 1111, 15000)}		
acc1	(0001, "Jack" 1111, 15000)	*Account_file*	{(0001, "Jack" 1111, 10000)}
amount	5000	*cash*	5000

Note that the animation only provides an example of executing the scenario but not capable of detecting any errors by itself. It is humans (e.g., the designer, the user, or anybody appointed as the inspector) who have to judge whether any errors have been revealed. The question is how can the animation be utilized to help humans uncover errors, especially those in relation to the user's requirements. Experience in practice suggests that software inspection can help to address the problem [75], to make the inspection more effective, specification animation can be utilized as a reading technique to guide the human to carry out the inspection. The problem is how the animation can be adopted to facilitate the inspection. We address this problem in the following chapters.

4.4 Summary

In this Chapter we illustrated the details of system functional scenario-based animation method. We first formally defined the system scenario, and then explained the algorithm for automatically deriving system scenarios based on the topology of CDFD. We also introduced the process for performing animation.

In the next Chapter, we will systematically introduce the traceability between the formal and informal specifications. The traceability rules between inspection targets and requirement items will be formally defined.

Chapter 5

Traceability of Specifications

Traceability between the formal specification and the informal specification is a measurement that tells how the corresponding parts in both specifications should be connected. The meaning of such a connection should reflect the idea that the requirements in the informal specification must be realized correctly in the formal specification. Before we utilize the traceability in forming appropriate checklist for inspection, we must figure out what parts in both specifications should be connected. This will allow us to trace the part of interest in a formal system scenario back to the corresponding part in the informal specification during inspection.

In this chapter, we discuss this issue by focusing on the four topics: explicit and implicit requirements, requirement items, inspection targets, and construction of traceability.

5.1 Explicit and Implicit Requirements

The connections between the informal and formal specifications mentioned above can be separated into two categories. The first kind of connection is called "*connection of explicit requirements*", which reflects the connections between the explicit requirement items in the informal specification and the corresponding parts in the formal specification. For convenience in discussion, we use the term "*explicit requirement items*" to mean the requirement items that are explicitly described in the informal specification, including functions, data resources, and constraints.

The other category of connection is called "*connection of implicit requirements*", which reflects the connections between the implicit requirement items in the informal specification and the corresponding realizations in the formal specification. Similarly, we use "*implicit requirement items*" to mean any requirement that is implicitly represented in the informal specification through relations between explicit requirement items. For instance, consider the two items: function *F1.1.2* and data resource *D2.1* in the informal specification shown in Figure 2.1 in Chapter 2. The

relationship between *F1.1.2* and *D2.1* is that function *F1.1.2* uses data item *D2.1.* The similar relationship also exists between function *F1.2.2* and data item *D2.1.* Such relationships should also be considered as requirements and realized properly in the formal specification.

5.2 Requirement Items

As stated previously, the requirements are divided into *explicit requirements* and *implicit requirements.* To make the requirements traceable, the categories of the requirements described in the informal specification must be formally defined. The explicit requirements are the requirements that are explicitly defined in the informal specification, including *functions*, *data resources*, and *constraints.* The following definition formally defines the explicit requirements in an informal specification.

Definition 3 *Let $S_I = (F_I, D_I, C_I)$ denote an informal specification, where F_I is the set of all function items, D_I is the set of all data resource items, and C_I is the set of all constraint items. Then, the explicit requirements of S_I are defined as follows, where "RQ" stands for a set of "requirements".*

$RQ_1 = F_I$

$RQ_2 = D_I$

$RQ_3 = C_I$

The implicit requirements are described by the relations between explicit requirements. The first kind of implicit requirement is the relation between the function items and the data resource items. This relation indicates what function items use what data resource items. A data resource item in the informal specification works like a database or file that provides necessary data to support the functionality of the desired system. Each data resource item described in the informal specification should be used by at least one function; otherwise the data resource item will be unnecessary for the system. The following function $using_D$ formally defines the relation between data resources and functions.

Definition 4 *Let the function $using_D$ be defined as:*

$using_D : D_I \rightarrow power(F_I)$

where $f \in using_D(d)$ if and only if function item f uses data resource item d. Then, the corresponding implicit

requirement is defined as follows:

$$RQ_4 = \{(d, f) \mid d \in D_I \wedge f \in F_I \wedge f \in using_D(d)\}$$

Applying the function $using_D$ to a data resource item d yields a set containing all the function items that use data resource item d. In this definition and the others to be given later in this dissertation, we use the symbol $power(a)$ to denote the power set of set a where a may be specialized to different set in different definitions.

For example, the informal specification of the simplified ATM system shown in Figure 2.1 contains nine functions and one data resource. The identifiers listed behind the data resource *D2.1* indicate that this data resource is used by four functions: *F1.1.2, F1.1.4, F1.2.2, and F1.2.3*. Therefore, the RQ_4 type implicit requirement includes four specific requirement items:

$$RQ_4 = \{(D2.1, F1.1.2), (D2.1, F1.1.4), (D2.1, F1.2.2), (D2.1, F1.2.3)\}$$

In addition to the RQ_4 type implicit requirement, the other two types of implicit requirements are the relations associating constraint items with function items or data resource items. We use $applyingto_D$ and $applyingto_F$ to represent these two relations and define them as follows.

Definition 5 *Let the two functions $applyingto_D$ and $applyingto_F$ be defined as:*

$$applyingto_D : C_I \rightarrow power(D_I)$$

$$applyingto_F : C_I \rightarrow power(F_I)$$

Then, the two types of implicit requirements are defined as follows:

$$RQ_5 = \{(c, d) \mid c \in C_I \wedge d \in D_I \wedge d \in applyingto_D(c)\}$$

$$RQ_6 = \{(c, f) \mid c \in C_I \wedge f \in F_I \wedge f \in applyingto_F(c)\}$$

Applying the function $applyingto_D$ to a constraint item c yields a set containing the data resource items that are restricted by constraint d, and applying the function $applyingto_F$ to a constraint item c yields a set containing the function items that are restricted by constraint d. Note that a constraint in the informal specification may restrict both function items and data resource items. Such constraint is usually the restriction to a specific kind of data, and the restricted data resources contain this kind of data and the restricted functions contain variables belonging

to this kind of data. For example, the constraints *C3.2* and *C3.3* in the informal specification in Figure 2.1 restrict both data resource *D2.1* and functions *F1.1.2* and *F1.2.2*. This is because these two constraints restrict the length of "*ID*" and "*password*" of a bank account, which are contained in *D2.1* and used by *F1.1.2*, *F1.2.2*. As parts of a bank account, the *ID* and *password* are contained in the data resource *D2.1* that is a set of bank accounts. In addition, the *ID* and *password* are used as input variables of functions *F1.1.2* and *F1.2.2* that describe the functionality of checking the password of the bank account with given ID.

The constraints that only restrict functions in the informal specification are usually the restrictions of the functionality of the system. For example, the *C3.4* declares that the maximum amount of withdraw is 10000. It is a part of the functionality of function "*Withdraw*", and should be formalized in the formal specification.

By analyzing the informal specification of the simplified ATM system, we conclude that the RQ_5 and RQ_6 types implicit requirements contains following requirement items:

$$RQ_5 = \{(C3.2,\ D2.1),\ (C3.3,\ D2.1)\}$$

$$RQ_6 = \{(C3.1, F1.1.1),\ (C3.1, F1.2.1),\ (C3.2,\ F1.1.2),\ (C3.2,$$

$$F1.2.2), (C3.3, F1.1.2), (C3.3, F1.2.2), (C3.4, F1.1.3)\}$$

5.3 Inspection Targets

In an inspection, the inspector is required to examine a specific part of the specification in each step of animation, namely, the part of the specification that is involved in the current step of animation. We call each specific content that needs to be inspected an *inspection target*. Similar to the requirement items in an informal specification, the inspection targets of a formal specification are also divided into two classes based on whether the inspection target is defined explicitly.

The first class of inspection target is the formal specification items defined explicitly, including *operation scenarios*, *state variables*, *type declarations*, and *invariants* that are defined in a module. As introduced previously, an operation scenario is a conjunction of *pre-condition*, *guard condition*, and *defining condition* that are extracted from the pre- and post-conditions of a process defined in the module; it defines an independent behavior in terms of input and output relation. A state variable corresponds to a data store in the associated CDFD, which is used as an external variable to provide necessary data to related process. It can be an independent variable, a file, or a

database in the implemented system. Similar to most programming language, each state variable in SOFL formal specification must be declared with a type before it can be used. If the type is not one of the basic type in SOFL, it has to be declared first. A type declaration is a statement of declaring a new name for a designated type in the module. An invariant is a logical expression that describes a constraint on either a state variable or a declared type. Such an invariant is required to be sustained throughout the entire system.

The second class of inspection target is the dependence relations between the formal specification items. Since different items defined in the formal specification work together to present the user's requirements, their dependence relations should also be inspected for their validity. All of these inspection targets are formally defined below; they will serve as a basis for forming questions in the checklist for inspection. We have developed a prototype tool to support the use of the inspection targets in forming the questions in the checklist.

Definition 6 *Let* $S_F = (M_1, M_2, ..., M_n)$ *denote a formal specification, where each* $M_i (i \in \{1, 2, ..., n\})$ *is a module defined in the specification.*

Definition 7 *Let* $M = (T_M, V_M, I_M, OS_M)$ *be a module in the formal specification* S_F*, where* T_M*,* V_M*,* I_M*,* OS_M *are the set of all type declarations, state variable declarations, invariants, and operation scenarios, respectively.*

Note that in above definition, the module contains the set of all operation scenarios derived from the processes in the module rather than the processes and the associated system scenarios themselves. This is because the operation scenario is the basic unit in the formal specification for presenting the desired functions, and both the process and system scenario can be transformed to a conjunction or disjunction of operation scenarios.

Definition 8 *Let* $OS_F = \bigcup_{i=1}^{n} M_i.OS_M$ *be the set of all operation scenarios defined in the entire formal specification,* $T_F = \bigcup_{i=1}^{n} M_i.T_M$*,* $V_F = \bigcup_{i=1}^{n} M_i.V_M$ *be the set of all type and state variable declarations respectively, and* $I_F = \bigcup_{i=1}^{n} M_i.I_M$ *be the set of all invariants. Then, the four kinds of inspection targets belonging to the first class are defined below, where "IT" stands for the set of "inspection target".*

$IT_1 = OS_F$

$IT_2 = V_F$

$IT_3 = T_F$

$IT_4 = I_F$

The second class of inspection target is dependence relations between different inspection targets defined in Definition 8. It describes how each part of a formal specification works together to represent the user's requirements. Inspecting dependence relations aims to check whether the collaborations between different formal specification items are correct and appropriate according to the user's requirements.

Based on the previous explanation of SOFL specifications, the definition of an operation scenario in the formal specification will involve the usage of state variables. Therefore, the basic inspection target *operation scenario* is directly involved in the dependence relations with *state variables*. For each state variable in the formal specification, it must be declared with a specific type. Thus, there are dependence relations existing between state variables and their types. In addition, since operation scenarios relate to state variables and state variables relate to their type, the dependence relations between operation scenarios and type declarations can be constructed through shared state variables. The following definition formally defines the three kinds of dependence relations described above and corresponding inspection targets.

Definition 9 *Let the following three functions present the dependence relations between operation scenarios and state variables, between state variables and types, and between operation scenarios and types, respectively.*

$using_V : V_F \rightarrow power(OS_F)$

$typeof_V : V_F \rightarrow T_F$

$typeof_{OS} : OS_F \rightarrow power(T_F)$

Then, the three kinds of inspection targets corresponding to the dependence relations are defined as:

$IT_5 = \{(v, os) \mid v \in V_F \wedge os \in OS_F \wedge os \in using_V(v)\}$

$IT_6 = \{(v, t) \mid v \in V_F \wedge t \in T_F \wedge t = typeof_V(v)\}$

$IT_7 = \{(os, t) \mid os \in OS_F \wedge t \in T_F \wedge t \in typeof_{OS}(os)\}$

The function $using_V$ indicates that a state variable can be accessed by more than one operation scenario. For example, the only state variable "*Account_file*" in the formal specification shown in Figure 2.2 in Chapter 3 is accessed by the operation scenarios defined in the three process "*Check_Password*", "*Withdraw*", and "*Show_Balance*". The function $typeof_V$ presents that each state variable belongs to a specific type; and the function $typeof_{OS}$ implies that an operation scenario may include more than one variable, therefore relate to more than one type declaration.

As explained above, each dependence relation is an inspection target and expressed as a pair of relevant inspection targets.

The invariant in the formal specification is a predicate that expresses a property to restrict a specific type. Therefore, a kind of dependence relation should exist to link the invariants and relevant types. The function that is used to describe this kind of dependence relation is formally defined as follows:

Definition 10 *Let $applying_I$ denote the dependence relation between invariants and types.*

$applying_I : I_F \rightarrow T_F$

Then, the inspection target corresponds to the dependence relation is defined as:

$IT_8 = \{(i, t) \mid i \in I_F \wedge t \in T_F \wedge t = applying_I(i)\}$

Since each type declaration may be involved in the dependence relations with operation scenarios or state variables, the invariants can be linked to operation scenarios or state variables through the types. These two kinds of dependence relations and corresponding inspection targets are formally defined as follows:

Definition 11 *Let the following functions present the dependence relations between invariants and operation scenarios, and between invariants and state variables, respectively.*

$applyingto_V : I_F \rightarrow power(V_F)$

$applyingto_{OS} : I_F \rightarrow power(OS_F)$

Then, the inspection targets corresponding to the dependence relations are defined as:

$IT_9 = \{(i, v) \mid i \in I_F \wedge v \in V_F \wedge v \in applyingto_V(i)\}$

$IT_{10} = \{(i, os) \mid i \in I_F \wedge os \in OS_F \wedge os \in applyingto_{OS}(i)\}$

As introduced in Section 3.1.3, the inspector checks all possible system scenarios derived from formal specification by following the animation process. In the inspection of each system scenario, the processes involved in the system scenario are inspected in turn. When inspecting a specific process, the basic inspection target to be checked is the involved operation scenario. It defines the unique functionality of the process that is involved in the system scenario. The operation scenario will lead the inspector to other related inspection targets based on the dependence chain shown in Figure 5.1. The oval nodes in Figure 5.1 represent inspection targets, and the arrows represent the

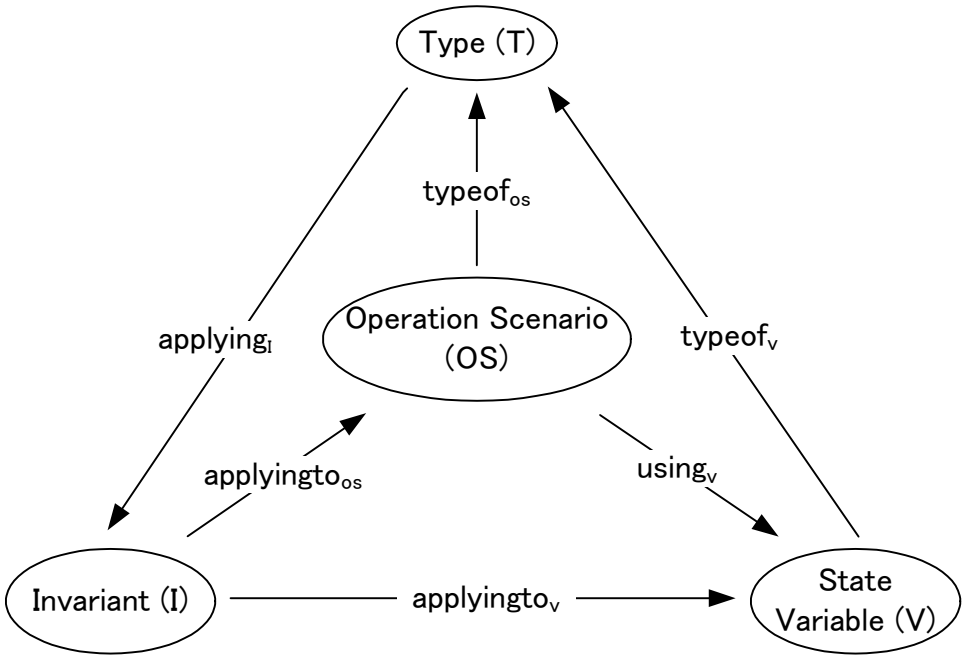


Figure 5.1: The dependence chain

dependence relations between different inspection targets, which have been formally defined previously. Starting from the operation scenario, all related inspection targets can be inspected by following the dependence chain.

For example, if system scenario {*withdraw_comm*} [*Receive_Command, Check_Password, Withdraw*] {*cash*} is under inspection, the inspection targets of each process will be checked in turn. Table 5.1 lists the inspection targets that need to be inspected.

The first column of Table 5.1 indicates the process that each inspection target belongs to. For example, the inspection target in the second row belongs to process *"Check_Password"*. The second column is the identifier of the inspection target, and the third column is its contents. The identifier "IT_1^2" consists of three parts: name *"IT"*, a subscript "1", and a superscript "2". The name "*IT*" stands for "Inspection Task" as we explained previously; the subscript "1" means it belongs to inspection target type IT_1; and the superscript "2" is its index. This inspection target is an operation scenario as indicated in the forth column of the table.

All of the inspection targets in Table 5.1 are derived from the process specifications shown in Figure 2.2, Figure 4.11, and Figure 4.12. Note that not all kinds of inspection targets defined above are included in the table. For example, the inspection targets in the second, forth and fifth rows are operation scenario IT_1^2, state variable IT_2^1

Table 5.1: The inspection targets of a specific system scenario

Process	**ID**	**Inspection Task**	**Description**
Receive_ *Command*	IT_1^1	*true and withdraw_comm* = "*withdraw*"$\wedge$ *sel* = *true*	operation scenario
Check_ Password	IT_1^2	*true and* (*exists*[*x* : *Account_file*] | *x.id* = *id and* *x.password* = *pass and sel* = *true and acc*1 = *x*)	operation scenario
	IT_2^1	*Account_file*	state variable
	IT_3^1	*Account*	type
	IT_3^2	*set of Account*	type
	IT_4^1	$forall[a : Account] \mid len(a.id) = 4$	invariant
	IT_4^2	$forall[a : Account] \mid len(a.password) = 4$	invariant
	IT_4^3	$forall[a, b : Account] \mid a <> ba.id <> b.id$	invariant
	IT_5^1	(IT_2^1, IT_1^2)	(variable, scenario)
	IT_6^1	(IT_2^1, IT_3^2)	(variable, type)
	IT_9^1	(IT_4^1, IT_2^1)	(invariant, variable)
	IT_9^2	(IT_4^2, IT_2^1)	(invariant, variable)
	IT_9^3	(IT_4^3, IT_2^1)	(invariant, variable)
	IT_{10}^1	(IT_4^1, IT_1^2)	(invariant, scenario)
	IT_{10}^2	(IT_4^2, IT_1^2)	(invariant, scenario)
	IT_{10}^3	(IT_4^3, IT_1^2)	(invariant, scenario)
Withdraw	IT_1^3	*true and amount* <= ˜*x.balance and x.balance* = ˜*x.balance* − *amount and cash* = *amount*	operation scenario

and type IT_3^2, respectively. Since the operation scenario IT_1^2 accesses the state variable IT_2^1 that is defined with type IT_3^2, the inspection target (IT_1^2, IT_3^2) should be included in Table 5.1 on the basis of Definition 9. However, it is not listed in the table since it does not present the user's most interests. By the inspection targets that present the user's most interests, we mean the inspection targets that can be traced back to the user's requirements based on the traceability rules defined in the next section.

5.4 Construction of Traceability

As mentioned previously, traceability between the formal specification and the informal specification is a measurement that describes how the corresponding parts in both specifications should be connected. Specifically, it shows the links from inspection targets in the formal specification to their original requirements in the informal specification. Each of such a connection is called a "*trace link*" in our inspection approach. To build the trace link, we must understand how each requirement item can be formalized; that is, we need to build the necessary relations between items in both specifications in general and then apply them as guidelines to assist the establishment of specific relations between the given informal and formal specifications when doing inspection for a specific specification.

The general corresponding relations between items in the informal and formal specifications are summarized in Table 5.2. The first type of requirement is function items. Ideally, each function item described in the informal specification is generally formalized and realized by one or more processes in the formal specification. Consider the informal specification in Figure 2.1 and the formal specification in Figures 2.2 and 2.3 as an example. The functions in the informal specification are organized in a two level hierarchy, while the processes in the formal specification are organized in only one level module. The function item $F1.1$ is not realized by any one of the processes in the formal specification. Actually, it is realized by a system scenario *{withdraw_comm}[Receive_Command, Check_Password, Withdraw]{cash}* instead. Since the function can be formalized by a process, an operation scenario, or a system scenario in the formal specification, it can be considered as realized by a group of operation scenarios in general.

Another type of requirement is data resource, which is usually realized by a state variable in the formal specification. In our inspection approach, we think the state variables and their types collaborate together to present

Table 5.2: Relations between specification items

Informal Specification	**Formal Specification**
function	process, operation scenario, system scenario
data resource	type declaration, state variable
constraint	invariant, process, operation scenario

the user's requirements about data resources, and the inspector has to check both the state variables and their types to make sure they satisfying the original requirements. For example, the only data resource in the informal specification is formalized by the only state variable "*Account_file*" in the formal specification. However, merely the "*Account_file*" is not enough for describing the data resource, its type "*set of Account*" must be considered collaboratively for explaining the data resource.

The constraints in the informal specification can be realized either by invariants or by part of the processes in the formal specification. In practice, the constraints on data resources are generally realized by invariants, and the constraints on functions are usually formalized as the pre- or guard conditions for restricting the functionality of the processes. For instance, the constraint *C3.1* in the informal specification requires that only two commands can be received by the simplified ATM system, and it restricts the functions *F1.1.1* and *F1.2.1*. In the formal specification, the constraint is formalized as two guard conditions in the process "*Receive_Command*". The guard conditions define the situations that can be handled by the process. Since only two guard conditions "*withdraw_comm = 'withdraw'*" and "*balance_comm = 'balance'*" are defined, only two commands can be handled in the process. Therefore the constraint *C3.1* is appropriately formalized in the formal specification.

Based on the general relations between the items in the informal and formal specifications, the trace links between inspection targets and requirement items can be constructed. Table 5.3 formally defines seven traceability rules. The rule $link_1$ in the first row of Table 5.3 formally describes the trace links between operation scenarios and functions. In this definition, $power(RQ_1)$ means the power set of RQ_1. It indicates that an operation scenario can realize one function, the combination of several functions, or a part of one function.

The rule $link_2$ in the second row of Table 5.3 formally describes the trace link from a pair of state variable and its type to a data resource. Since the inspector has to check both the state variable and its type to make sure a

Table 5.3: Traceability rules between inspection targets and requirement items

ID	Inspection Target (Description)	Requirement (Description)	Definition
$link_1$	IT_1 (operation scenario)	RQ_1 (function)	$link_1 : IT_1 \rightarrow power(RQ_1)$
$link_2$	IT_6 ((state variable, type))	RQ_2 (data resource)	$link_2 : IT_6 \rightarrow RQ_2$
$link_3$	IT_4 (invariant)	RQ_3 (constraint)	$link_3 : IT_4 \rightarrow RQ_3$
$link_4$	IT_1 (operation scenario)	RQ_3 (constraint)	$link_4 : IT_1 \rightarrow RQ_3$
$link_5$	IT_5 ((state variable, operation scenario))	RQ_4 ((data resource, function))	$link_5 : IT_5 \rightarrow RQ_4$
$link_6$	IT_9 ((invariant, state variable))	RQ_5 ((constraint, data resource))	$link_6 : IT_9 \rightarrow RQ_5$
$link_7$	IT_{10} ((invariant, operation scenario))	RQ_6 ((constraint, function))	$link_7 : IT_{10} \rightarrow RQ_6$

data resource is appropriately realized in the formal specification, we do not construct the trace link between a state variable and a data resources.

The traceability rules $link_3$ and $link_4$ define how an inspection target can be trace back to the constraint in the informal specification. The two kinds of inspection targets that can link to constraints are invariants and operation scenarios. As explained earlier, the invariants link to the constraints that restrict data resources, and the operation scenarios link to the constraints restricting functions.

In addition to the explicit requirement items, the three types of implicit requirements should also be formalized in the formal specification. Since the implicit requirements describe the relations between different explicit requirements, their formalization is usually realized in the formal specification as the dependence relations between the corresponding inspection targets. For example, one of the implicit requirements in the informal specification shown in Figure 2.1 in Chapter 2 is that function *F1.1.2* access data resource *D2.1*. Since the two explicit requirement items *F1.1.2* and *D2.1* are formalized by the inspection targets IT_1^2 and IT_2^1 in Table 5.1, respectively, the implicit requirement item is realized by the dependence relation between IT_1^2 and IT_2^1. If the two inspection targets are formalized appropriately and their dependence relation does exist in the formal specification, we can conclude

Table 5.4: The trace links between the inspection targets and the requirement items

No.	Inspection Task	Requirement Item	Rule
1	IT_1^1 (operation scenario)	$F1.1.1$ (function)	$link_1$
2	IT_1^2 (operation scenario)	$F1.1.2$ (function)	$link_1$
3	IT_4^1 (invariant)	$C3.2$ (constraint)	$link_3$
4	IT_4^2 (invariant)	$C3.3$ (constraint)	$link_3$
5	IT_5^1 (state variable, operation scenario)	($D2.1$, $F1.1.2$) (data resource, function)	$link_5$
6	IT_6^1 (state variable, type)	$D2.1$ (data resource)	$link_2$
7	IT_9^1 (invariant, state variable)	($C3.2$, $D2.1$) (constraint, data resource)	$link_6$
8	IT_9^2 (invariant, state variable)	($C3.3$, $D2.1$) (constraint, data resource)	$link_6$
9	IT_{10}^1 (invariant, operation scenario)	($C3.2$, $F1.1.2$) (constraint, function)	$link_6$
10	IT_{10}^3 (invariant, operation scenario)	($C3.3$, $F1.1.2$) (constraint, function)	$link_7$
11	IT_1^3 (operation scenario)	$\{F1.1.3, F1.1.4\}$ (function)	$link_7$

that the implicit requirement is appropriately formalized; otherwise, it is not appropriately formalized. The traceability rules $link_5$, $link_6$, and $link_7$ in Table 5.3 describe the trace links from different dependence relations to corresponding implicit requirements.

It should be noted that according to Table 5.3, not all of the inspection targets defined in Chapter 5.3 can be directly linked to a requirement item. For example, the inspection target "*type declaration*" is not linked to any of the requirements. However, the dependence relations involving type declarations, such as the relations between state variables and types, are treated as inspection targets and directly linked back to the requirements in the formal specification. When inspecting such dependence relations, the type declarations themselves can be inspected. In the formal specification, each item is defined for formalizing the desired requirements directly or indirectly. In the inspection, every item defined in the formal specification should be considered as an inspection target and inspected for validation even if it cannot be linked to a specific requirement item.

Table 5.4 lists up the trace links between the inspection targets in Table 5.1 and the requirement items mentioned in Chapter 5.2. Those inspection targets that cannot be linked to a specific requirement item are not listed in the

table.

5.5 Summary

In this Chapter, we discussed the traceability between the informal and formal specifications and the relations between different items defined inside specifications. The explicit and implicit requirement items were formally defined. The inspection targets of the formal specification were also formally defined. Based on the formal definitions of the requirement items and inspection targets, we proposed a group of traceability rules for building trace links. In the next Chapter, we will explain how a checklist can be constructed based on the traceability, and how the internal dependence relations can guide the inspector check related inspection targets.

Chapter 6

Formal Specification Inspection using IB-SAT

The inspection approach proposed in this dissertation is based on the animation method and the traceability introduced in the previous chapters. In this chapter, we explain how to coordinate the animation and traceability to inspect a specification. In the first section, we propose four aspects for each inspection target. These aspects should be checked in the inspection. Then we introduce how to construct checklist based on the traceability and explain how to systematically inspect all inspection targets. We also discuss how to modify the specification based on the results of inspection. Finally, we use a case study to demonstrate the entire inspection process at the end of this chapter.

6.1 The Four Aspects of Inspection Targets

In order to check whether a formal specification accurately and appropriately formalizes the user's requirements, each inspection target defined in the formal specification should be inspected from four aspects: *necessity*, *appropriateness*, *correctness*, and *completeness*. These four aspects are defined as four basic properties of the formal specification. We explain the meanings of these four aspects and why they need to be inspected. Some satisfactions of the aspects can be formally defined and some others cannot. For the satisfactions that can be formally defined, specific properties are defined as predicate expressions and the inspector can apply related information to the specific properties for defining rigorous inspection tasks. The inspection task can give the inspector specific instruction for making decision whether a specific property is satisfied. For those satisfactions that cannot be formally defined, questions lacking rigorous are asked and the inspector needs answer these questions to judge

whether the basic properties are satisfied.

6.1.1 Necessity

Defining unnecessary items in the formal specification will affect the readability and maintainability of the specifications. Moreover, the unnecessary formal specification items would confuse its user, such as the programmers who implement the system based on the formal specification.

Checking the necessity property aims to ensure that no declared type identifier, variable, or invariant is not used in the process or function specifications of each module. For example, if a declared type identifier is never used for declaring any variable in the specification, the declaration of the type identifier will be regarded as unnecessary. The similar principle can be applied to variables and invariants in modules.

Take the advantage of the formalization of inspection targets, the necessity property can be formally defined as follows:

Definition 12 ***Property 1*** *All the items defined in the formal specification are considered necessary if the following three conditions hold:*

$1)\forall_{t\in T_F}((\exists_{os\in OS_F}\cdot t \in typeof_{OS}(os)) \vee (\exists_{v\in V_F}\cdot t = typeof_V(v)))$

$2)\forall_{v\in V_F}\exists_{os\in OS_F}\cdot os \in using_V(v)$

$3)\forall_{i\in I_F}\exists_{t\in T_F}\cdot t = applying_I(i)$

The first condition in the definition requires that all types defined in the formal specification should be used by at least one operation scenario or declared as the type of a state variable. The two functions "$typeof_{OS}$" and "$typeof_V$" defined in Chapter 5.3 present the dependence relations of type declarations with operation scenario and state variable, respectively. If the three conditions are satisfied, it implies that all the type declarations, state variables, and invariants defined in the formal specification are necessary. Note that the necessity of processes is not checked here. This is because the inspector cannot decide whether a process is necessary without considering the user's requirements described in the informal specification.

If the above property is not satisfied, we can conclude that some formal specification items are defined inappropriately, and the inspector should consider why some defined items are not used. However, even if the property

is satisfied, it does not mean that all these declarations are correct and appropriate. This property is only a prerequisite for further inspection. In our inspection approach, we adopt this property as a guideline to check even further on the appropriateness of the variable and type declarations as well as invariants with respect to the informal requirements.

6.1.2 Appropriateness

Appropriateness requires that the inspection targets realize the original requirements in an appropriate manner. The appropriateness of an inspection target usually cannot be formally defined. The inspector needs use his own knowledge and experience to decide whether the inspection target is formalized appropriately. In general, we think an inspection target is appropriate if it intuitively presents all necessary information of the original requirement.

For example, the state variable "*Account_file*" is defined in the formal specification shown in Figure 2.2 with a type "*set of Account*". This state variable and its type realize the data resource *D2.1* described in the informal specification shown in Figure 2.1. To check the appropriateness of the inspection target "(*Account_file, set of Account*)", the inspector needs to answer some questions, such as "Whether the name of the state variable '*Account_file'* represents the essence of the data resource D2.1?" and "Whether the type '*set of Account'* of the state variable is appropriate for the data resource D2.1?". The answers of these questions are highly dependent on the inspector's judgement. Some inspector may think "*Account_file*" is a good name for the state variable, some inspector may think the name "*Account_database*" is more appropriate. Involving personal bias is inevitable when an inspector answers the appropriateness questions.

Note that the original requirements of an inspection target will appear in the questions to help the inspector understand the questions and make decisions. In previous example, the appearance of data resource D2.1 in the questions provide the inspector with necessary information to make judgement. Comparing to a more abstract question "Whether the name of the state variable is appropriate?" or "Whether the state variable is defined appropriately?", the questions including corresponding requirements information are more specific. To construct questions containing related requirement items for inspection targets, the trace links between the formal and informal specifications must be utilized. In Chapter 5.4, the rules for building trace links are formally defined. However, the procedure for constructing the trace links based on these rules is not given. In our inspection

approach, the inspector constructs the trace links for each inspection target by answering several traceability related questions. Then, the specific questions that contain related requirement items can be asked based on the trace links for each inspection targets. The traceability related questions will be introduced in the next section.

The questions about the inspection targets whose appropriateness cannot be formally defined are usually asked based on inspector's experience. The experience generally comes from the previous development projects that the inspector has participated in. Table 6.1 lists some typical appropriateness questions based on our experience, but appropriateness questions are not limited to the questions listed in the table.

For the three kinds of inspection targets whose appropriateness can be formally defined, properties are formally defined to help the inspector to make judgement. The three kinds of inspection targets are the dependence relations that can be linked to the implicit requirement items in the informal specification, namely, the inspection targets *(state variable, operation scenario)*, *(invariant, state variable)*, and *(invariant, operation scenario)*. The corresponding implicit requirement items are *(data resource, function)*, *(constraint, data resource)*, and *(constraint, function)*, respectively.

The reason that the appropriateness of these three kinds of inspection targets can be formally defined is that the corresponding implicit requirement items can be formally defined. Considering the three inspection targets in Table 6.1, they are linked to the explicit requirement items, which are described by informal natural languages in the informal specification. In order to inspect the appropriateness of the three kinds of inspection targets in Table 6.1, the inspector has to read the corresponding informal descriptions, understand the descriptions, and then makes the judgement based on his knowledge. On the contrary, the implicit requirements in the informal specification can be formally defined. Checking the appropriateness of the inspection targets that formalize the implicit requirements is equivalent to check whether the dependence relations between different inspection targets are consistent with the relations between corresponding explicit requirements.

For example, the inspection target T_9^1 in Table 5.4 in Chapter 5.4 is the dependence relation between an invariant and a state variable. To check the appropriateness of this inspection target is actually to check whether the dependence relation is appropriate, namely, whether the invariant should be used to restrict the state variable. This question can be addressed by considering its original requirement. Based on the trace link listed in Table 5.4, the original requirement is an implicit requirement that represents the relation between a constraint and a data

Table 6.1: Typical appropriateness questions for different inspection targets

Inspection Target	**Requirements**	**Questions**
(state variable, type) (IT_6)	data resource (D)	Whether the name of state variable represents the essential idea of the data resource?
		Whether the type of state variable represents the essential idea of the data resource?
		Whether the name is defined without ambiguity?
invariant (IT_4)	constraint (C)	Whether the quantifier used in the invariant reflects the essential of the constraint?
		Whether the type restricted by the invariant complies with the constraint?
		Whether the content of the invariant represents the essential idea of the constraint?
operation scenario (IT_1)	function (F)	Whether the name of process represents the essential idea of the function?
		Whether the input variables of the operation scenario are necessary for the function?
		Whether the names of types of input variables are defined appropriately?
		Whether the output variables of the operation scenario are necessary for the function?
		Whether the names of types of output variables are defined appropriately?
		Whether the pre-condition of the operation scenario presents the prerequisite of the function?
		Whether the guard condition of the operation scenario precisely define the situation of the function?

resource. If the dependence relation represented by the inspection target is consistent with the relation presented by the original implicit requirement, the inspection target can be considered appropriate. Therefore, the inspector should check whether the invariant and state variable in the inspection target formalize the constraint and data resource in the implicit requirement, respectively. If the answer is "yes", that means the two relations are consistent and the inspection target is appropriate. Otherwise, the inspection target is defined inappropriately.

The appropriateness of inspection target *(invariant, state variable)* is formally defined as follows:

Definition 13 ***Property 2*** *If the dependence relation of IT_5 is appropriate, the following predicate must hold:*

$$\forall_{v\in V_F, os\in OS_F, d\in D_I, f\in F_I} \cdot(d,f) = link_5((v,os)) \Rightarrow$$

$$\exists_{t\in T_F} \cdot(t = typeof_V(v) \wedge d = link_2((v,t)))\wedge$$

$$f \in link_1(os) \wedge os \in using_V(v) \wedge f \in using_D(d)$$

The property states that if there is a trace link linking inspection target *(v, os)* to requirement *(d, f)*, the state variable v must realize data resource d and the operation scenario os must realize function *f*.

Similarly, the dependence relations of inspection targets *(state variable, operation scenario)* and *(invariant, operation scenario)* should satisfy the following two properties for their appropriateness.

Definition 14 ***Property 3*** *If the dependence relation of IT_9 is appropriate, the following predicate must hold:*

$$\forall_{i\in I_F, v\in V_F, c\in C_I, d\in D_I} \cdot(c,d) = link_6((i,v)) \Rightarrow$$

$$c \in link_3(i) \wedge \exists_{t\in T_F} \cdot(t = typeof_V(v) \wedge d = link_2((v,t)))\wedge$$

$$v \in applyingto_V(i) \wedge d \in applyingto_D(c)$$

Definition 15 ***Property 4*** *If the dependence relation of IT_{10} is appropriate, the following condition must hold:*

$$\forall_{i\in I_F, os\in OS_F, c\in C_I, d\in D_I} \cdot(c,f) = link_5((i,os)) \Rightarrow c \in link_3(i)$$

$$\wedge f \in link_1(os) \wedge os \in applyingto_{OS}(i) \wedge f \in applyingto_F(c)$$

The formally defined properties may make automation of the inspection for appropriateness becomes possible. For example, the Property 2, 3, and 4 can be checked automatically by program as longs as the related trace links between inspection targets and explicit requirement items are built. However, the automation is based on the assumption that the trace links between related inspection targets and explicit requirements are constructed

correctly. For instance, in previous example, the appropriateness of inspection target T_9^1 is judged by applying the property defined in Definition 13. But, the conclusion is correct if and only if the trace links of the invariant and state variable are constructed correctly. If the invariant is linked to an incorrect constraint, the conclusion would be that the inspection target T_9^1 is defined inappropriately, even it does reflect the original requirement. Actually, whether the trace link between an explicit defined inspection target and its corresponding requirement item is constructed correctly can be verified by answering related appropriateness questions. Since the questions are raised based on the trace link, the negative answers of the appropriateness questions may reveal that the trace link is built incorrectly. The details can be found in the following sections.

6.1.3 Correctness

We use "correctness" as a property of an inspection target, requiring that the target satisfies both the syntax of the formal specification language and the two properties given in Definitions 16 and 17 below.

Definition 16 ***Property 5*** *Let $P_{pre} \wedge C_i \wedge D_i$ be an operation scenario, where P_{pre} is the pre-condition, C_i is guard condition, and D_i is defining condition. Then the following two predicates must hold:*

1)$\forall_{x,\sim s} \cdot P_{pre}(x, \sim s) \Rightarrow \exists_i \cdot C_i(x, \sim s)$

2)$\forall_{x,\sim s} \cdot (P_{pre}(x, \sim s) \wedge C_i(x, \sim s)) \Rightarrow \exists_{y,s} \cdot D_i(x, y, \sim s, s)$

In this definition, x and y are the set of input and output variables respectively. The decorated state variable $\sim s$ denotes the value of s before the execution of the operation scenario. The property states that for every input satisfying the pre-condition of the process P, there must exist a guard condition C_i satisfied by the same input and for every input satisfying both the pre-condition and the guard condition C_i, there must exist an output satisfying the corresponding defining condition D_i.

This property is also known as satisfiability proof obligation in the literature and it is used to check the internal consistency of an operation scenario. In the inspection, each operation scenario must be inspected to ensure the internal consistency. For example, the inspection target T_1^3 in Table 5.1 is an operation scenario of process "*Withdraw*". By applying Property 5 to this operation scenario, the following two predicate expressions can be generated:

1) $true \Rightarrow amount <= \tilde{}x.balance$

2) $true \wedge amount <= \check{}x.balance \Rightarrow\ x.balance = \check{}x.balance - amount \wedge cash = amount$

The above predicates can be automatically formed with tool support, but its formal proof is usually challenging. In our approach, such a property is checked by means of inspection.

Definition 17 ***Property 6*** *Let one of the related invariants on type T be $I_t = \forall_{t \in T} \cdot Q(t, w)$. Then the following predicates must hold:*

1)$P_{pre}(v) \Rightarrow Q(t, w)[v/t]$

2)$\check{}P_{pre}(v) \wedge C_i(v) \wedge D_i(v) \Rightarrow Q(t, w)[v/t]$

Since the invariant and the operation scenario are defined separately, only performing the syntax checking may not ensure that the relevant variables in the operation scenario comply with the related invariant. Abstractly, this property states that the type invariant must hold before and after the execution of the operation scenario $\check{}P_{pre}(v) \wedge C_i(v) \wedge D_i(v)$. Specifically, it requires two things. One is that when the pre-condition involving variable v of type T holds before the execution of the operation scenario, the invariant predicate $Q(t, w)$ must also be satisfied by variable v after substituting v for variable t in the predicate. The other is that if the conjunction of the guard condition and the defining condition involving variable v holds, it must guarantee the invariant predicate $Q(t, w)$ to be true after the variable substitution.

For example, the inspection target T_1^2 in Table 5.1 is an operation scenario of process "*Check_Password*" and inspection target T_4^1 is an invariant of type "*Account*". By applying Property 6, the following two predicates can be constructed:

1) $true \Rightarrow forall[a : Account\_file] \mid len(a.id) = 4$

2) $true\ and\ (exists[x : Account\_file] \mid x.id = id\ and\ x.password = pass)$

$$and\ sel = true\ and\ acc1 = x \Rightarrow len(x.id) = 4$$

Note that the invariant is defined as a restriction to a specific type, in Property 6, it is equivalent to the restrictions to all the variables that are defined with the specific type. The bound variable x in above predicate is with type "*Account*" and therefore restricted by the invariant. Since variable x is explicitly used in the operation scenario, it can be directly used in the invariant. For those restricted variables that are implicitly used in the operation scenario as a part of compound variables, they must be extracted first before they can be used in the

Table 6.2: Extracting restricted variables from the variables with compound type

Compound Type	**Definition**	**Expression**
Set	*var*: set of T	$forall[x : var] \mid Q(x)$
Sequence	*var*: seq of T	$forall[x : inds(var)] \mid Q(var(x))$
Composed	*var* = composed of *x*: T .... end	$Q(var.x)$
Product	*var*: T * T_1 * T_2 ...	$Q(var(1))$
Map	*var*: map T to T_1	$forall[x : dom(var)] \mid Q(x)$
Union	*var* = $T \mid T_1 \mid T_2 \mid ... \mid T_n$	$forall[x : var.is\_T(x)] \mid Q(x)$
Class	class A *x*: T .... end_class *var*: A	$Q(var.x)$

invariant. By compound variable, we mean the variables defined with *compound types*. The compound types in SOFL formal specification are some kinds of integration of basic types for presenting more complex data structures. For variables with different compound types, different methods are used for extracting restricted variables.

Table 6.2 shows some possible ways to extract restricted variables from compound type variables [93]. In this table, *var* is the name of the compound type variable, T is the type restricted by invariant, $Q(v)$ is a predicate that presents the invariant of variable v. In addition, *inds* is an operator of sequence type, it returns the set of indexes of the sequence operand; *dom* is an operator of map type, and it returns domain of the map type variable; $is\_T(x)$ is a method of type union, and it is used to check whether the parameter x is in type T.

Moreover, the correctness requires the inspection targets comply with customized standards or criteria. For example, a development team or software company may ask the external variables in the formal specification to be named in *Pascal* style and the local variables to be named in *Camel* style. The correctness of an inspection target requires such standard must be satisfied. Since the inspection targets are formally defined in our approach, inspector can use templates to easily construct specific questions. The customized questions will significantly extent the capability of our inspection approach.

6.1.4 Completeness

Checking the completeness of the formal specification is to examine whether every user's requirement written in the informal specification is defined properly in the formal specification. Since there is no restriction that each requirement item in the informal specification must be formalized by one specific formal specification item, the relations between requirement items and inspection targets may not be one-to-one relation. Some requirement items (e.g., data resources or constraints) in the informal specification may be represented by one item in the formal specification, while other requirement items (e.g., functions) may be realized by more than one item in the formal specification.

For the requirement item that can be formalized by one specific inspection target, its completeness can be determined by considering the appropriateness and correctness of the related inspection target. For the requirement item that may be realized by more than one inspection target, the inspector can hardly make any judgement of completeness by merely inspecting each related inspection target independently. All related inspection targets must be inspected as a whole to ensure the requirement item is formalized properly.

As indicated by the traceability rules in Table 5.3 in Chapter 5.4, the only requirement item that may be formalized by more than one inspection target is the function item. This situation may be caused by two reasons. One is that the function is a high level function in the informal specification, and it is decomposed into several related lower level functions. In the formal specification, the lower level functions are realized by different operation scenarios, therefore, the high level function is realized by the combination of a group of operation scenarios. The other reason is that an independent lower level function in the informal specification is decomposed in the process of formalization for refinement, so that the functionality of this lower level function is formalized by several operation

scenarios in the formal specification.

To check the completeness of the functions that realized by more than one operation scenario, the operation scenarios realizing the same function item should be well organized for inspection. The following disjunctive normal form presents the appropriate form to organize the related operation scenarios of a specific function:

$$(os_1^1 \wedge os_2^1 \wedge ... \wedge os_x^1) \vee (os_1^2 \wedge os_2^2 \wedge ... \wedge os_y^2)$$

$$\vee ... \vee (os_1^m \wedge os_2^m \wedge ... \wedge os_z^m)$$

In this disjunction, all of the operation scenarios, os_q^p, can be traced to the same function item in the informal specification. The operation scenarios in each conjunction clause belong to the same system scenario because the operation scenarios in the same system scenario work together to describe the functionality. The conjunction clauses are connected disjunctively since different system scenarios are mutually exclusive. By inspecting the disjunction as a whole, the inspector can judge whether a function item is realized completely.

For instance, the function *F1.1.2,* "Check password", is an independent lower level function in the informal specification shown in Figure 2.1 in Chapter 2. It decomposed into two operation scenarios in the formal specification. To check whether the function *F1.1.2* is formalized completely, the two operation scenarios must be reorganized into the format mentioned above for inspection. The reorganized disjunction is shown as follows:

true and exists![x: Account_file] | (x.id = id and x.password = pass and sel = true and acc1 = x)

$\vee$

true and not(exists![x: Account_file] | (x.id = id and x.password = pass)) and err1 = "Reenter your password or insert the correct card"

These two operation scenarios are connected disjunctively since they belong to two different system scenarios. The inspector should use his own experience and skill to determine whether the above disjunction completely describes the function "Check password".

The function *F1.1,* "Withdraw", is a high level function in the informal specification, and it is decomposed into four lower level functions. In the formal specification, the function *F1.1* is not formalized by a specific operation scenario. Instead, it is realized by the group of operation scenarios that formalize its four lower level functions. The following disjunction should be inspected to ensure that the function "Withdraw" is completely formalized.

true and withdraw_comm = "withdraw" and sel = true ∧

true and exists![x: Account_file] | (x.id = id and x.password = pass and sel = true and acc1 = x)

∧ true and amount <= ˜x.balance and x.balance = ˜x.balance - amount and cash = amount

∨

true and withdraw_comm = "withdraw" and sel = true ∧ true and not(exists![x: Account_file] | (x.id = id and x.password = pass)) and err1 = "Reenter your password or insert the correct card"

The two conjunction clauses in above disjunction present two different system scenarios: {*withdraw_comm*} [*Receive_Command, Check_Password, Withdraw*] {*cash*} and {*withdraw_comm*} [*Receive_Command, Check_Password*] {*err1*}. That means the function "Withdraw" is actually formalized by two system scenarios in the formal specification.

6.2 Checklist and Inspection Procedure

As indicated in Figure 4.13 in Chapter 4, when the inspector reads through the formal specification by following the animation steps, he can focus on inspecting a group of inspection targets related to a specific process. The basic inspection target to be checked in the inspection of a specific process is operation scenario, which defines the unique functionality of the process involved in the system scenario. Other inspection targets related to the operation scenario can be checked by following the dependence chain shown in Figure 5.1 in Chapter 5.3.

To inspect each inspection target, the inspector should answer several questions raised from the traceability aspect and the four aspects mentioned in the previous section. Answering the traceability related questions not only checks the inconsistency between the informal and formal specifications, but also helps the inspector build the trace links between inspection targets and corresponding requirement items. The constructed trace links will contribute to raise other questions, e.g. the questions based on the properties defined in Definition 13, 14, or 15. Table 6.3 lists up the typical traceability related questions for each explicitly defined inspection target.

In addition to the questions related to the traceability, the inspector also needs to answer the questions raised from the four aspects of the inspection targets to ensure that every property is satisfied. For example, according to the property defined in Definition 12, the inspector should answer the question "Whether is it necessary?" for checking the necessity of each explicitly defined formal specification item. If an item is used somewhere else in the

Table 6.3: Checklist for inspection

Items	**Questions**
type declaration	Does this type declaration define a type for any data resource? What is the data resource?
variable declaration	Does this state variable realize any data resource? What is the data resource? Whether the variable and its type properly formalize the data resource?
invariant definition	Does this invariant realize any constraint? What is the constraint?
operation scenario	Does this operation scenario realize any function? What is the function? Does the function use any data resource? Does this operation scenario use the variable that formalizes the data resource? Does the function comply with any constraint? Does the operation scenario satisfy the invariant that formalizes the constraint or does this operation scenario realize the constraint itself?

formal specification, then the answer is "Yes", otherwise, the answer is "No". In the program, defining an item that is never used may not be an error as long as the program can be executed correctly. However, a never used item may indicate some potential errors contained in the formal specification. The errors may be the incorrect definition of the item, or the incorrect usage of an item that is defined correctly. The inspector should carry out further inspection to reveal the reason that causes the situation.

Based on the trace links, the questions asked from appropriateness can be specific. By specific, we mean that each appropriateness related question contains specific inspection target and corresponding requirement item. For example, an abstract appropriateness related question may be "Whether the name of each process represents the essence of the user's requirements?". This question is suitable for the inspection of the entire formal specification. However, when answering this question, the inspector may ask himself "Where is the process?", "Where is the corresponding user's requirements?", etc. In our inspection approach, animation will guide the inspector to each process, and the trace links can be adopted to ask a specific question for each process. Assuming a process named "P" contains an operation scenario "OS" that formalizes a function called "F" in the informal specification. Based

on the trace link between "OS" and "F", a specific question can be constructed: "Whether the name of process 'P' that contains operation scenario 'OS' represents the essence of function 'F'?". Obviously, the specific question provides the inspector with more information for making judgement. Since the inspection targets, requirement items, and trace links are formally defined, automatically constructing specific questions for each explicit defined inspection target is possible with tool support.

Moreover, the appropriateness about the inspection targets that presents the dependence relations between different formal specification items can be automatically checked based on the properties defined in Definition 13, 14, and 15. But the correctness of the results is highly dependent on whether the related trace links between the explicitly defined inspection targets and requirement items can be constructed correctly. For instance, if the trace links between operation scenarios and functions can not be constructed correctly, the appropriateness of inspection targets IT_5 ((state variable, operation scenario)) and IT_{10} ((invariant, operation scenario)) cannot be correctly checked.

Actually, the appropriateness and correctness related questions will impel the inspector to verify the trace links built by answering the traceability related questions. After the inspection, the correct trace links between formal specification items and the requirement items can be constructed. As indicated by existing research [94] [95], the trace link itself is considered as an important approach for ensuring the quality of the requirement specifications.

To check the correctness of each inspection targets, the inspector should ensure that each of them complies with the syntax of SOFL and the properties defined in Definition 16 and 17 are satisfied. Therefore, for each operation scenario, the question "Whether the operation scenario is defined complying with the SOFL syntax?", "Whether the operation scenario satisfies the Property 5?", and "Whether the operation scenario satisfies the Property 6?" will be asked. As demonstrated in the previous section, answering the later two questions is actually checking the predicates that are generated by applying the corresponding properties to the operation scenario. In our inspection method, the inspector can use test suites to evaluate the predicates rather than to formally prove them. A test suite for an operation scenario contains the test case and expected result. By applying the test case and expected result to the predicates, the inspector can evaluate each predicate to be *true* or *false*. If the result of evaluation is *false*, it indicates that the corresponding property cannot be satisfied and the inspector needs to modify and refine the formal specification to keep its internal consistency. This is because the test suite presents a specific circumstance

that the property cannot be satisfied. Note that evaluating a predicate using test suites is not equivalent to a formal proof; it can only tell whether the predicate is satisfied or not on the sample data provided by the test suite. In spite of this inferiority, testing-based evaluation can enjoy advantage over formal proof in providing full automation.

The completeness related questions will be asked after all inspection targets are inspected. As explained previously, all the inspection targets that formalize the same requirement item must be inspected as a whole to ensure the completeness of the requirement item. The trace links constructed in the inspection of each individual inspection target will be used to raise the questions.

6.3 Feedback

The feedback of the inspection is actually the answers to the questions on the checklist. The designer should modify the formal specification based on the feedback. According to the feedback of different question categories, the designer may modify the specification from different aspects. For example, two kinds feedback of traceability-related questions will lead to modifications. One is that a requirement item is not formalized in the formal specification, and the other is that a formal specification item defines what is not described in the informal specification. For the former, the designer needs to consider whether a new formal specification item should be defined to formalize the requirement item. For the latter, the designer should consider whether the definition of the formal specification item is necessary.

One of the possible actions to respond to the feedback of necessity-related question is to delete the formal specification item that is not used anywhere else in the formal specification. However, the designer should not delete it directly since merely referring to the feedback of necessity-related question may not indicate precisely whether the item is unnecessary or forgotten to be used. The designer should not make any action to modify the formal specification until all the concerns in the feedback are addressed.

The designer should consider more when responding to the feedback of the appropriate-related questions. Since the appropriateness of the explicitly defined inspection targets cannot be formally defined as properties, whether the inspection targets are appropriate is judge by the inspector based on his own experience. Before the designer does any modification, he needs first discuss with the inspector who thinks an inspection target is inappropriate.

The inspector should explain the reason why he makes such result to the designer. If the designer is in agreement with the inspector, he can make the modifications based on the discussion. Otherwise, they should make further discussion until all the disagreements have been eliminated. For the inspection targets whose appropriateness can be formally defined as properties, if the related properties cannot be satisfied, the designer needs make modifications to ensure that all these properties are satisfied.

The feedback of a correctness-related question can be "correct" or "incorrect". In our definition, the incorrect result indicates that the definition of the inspection target is syntactical incorrect or contains internal inconsistency. The specification must be corrected before further inspection can be carried out. Even there is no other problem indicated by the feedback of the inspection, a formal specification containing incorrect item is useless in practice. The feedback of completeness-related questions may reveal that a specific requirement item is not formalized completely in the formal specification. In this situation, the designer needs to consider what kind of formal specification items should be added to the specification so that the formalization of the requirement item can be complete.

6.4 Case Study

This section presents a case study we have conducted to demonstrate how our inspection method works in practice. Our focus is on the explanation of how the inspection approach can be applied to a concrete formal specification for validation and verification rather than on its systematic evaluation.

We choose the simplified ATM software system as the target system for this case study. Figures 2.2 and 2.3 in Chapter 2 show the selected formal specification in our case study. The corresponding requirements are documented in the informal specification as shown in Figure 2.1.

To inspect the formal specification, a checklist is constructed based on both the general rules for trace links of the traceability between the formal and informal specifications and the general definitions of the four aspects of formal specifications. Based on the characteristics of each category of questions, the entire inspection process is separated into three stages. In the first stage, the inspector examines the formal specification to check whether all of the defined items are used anywhere else in the specification, which is required by the demand for inspecting the "necessity" property defined previously. The second stage is to inspect the formal specification by following

Table 6.4: The results of inspecting the necessity property

Formal Spec Item	**Category**	**Used By**	**Category**	**Condition**
Account	Type	#Account_file	Variable	1
		Check_Password	Process	1
		Withdraw	Process	1
		Show_Balance	Process	1
#Account_file	Variable	Check_Password	Process	2
		Withdraw	Process	2
forall[a: Account] | len(a.id) = 4	Invariant	Account	Type	3
forall[a: Account] | len(a.password) = 4	Invariant	Account	Type	3
forall[a, b: Account] | a <> b => a.id <> b.id	Invariant	Account	Type	3

the animation process. At this stage, the inspector should answer the questions raised based on the traceability, appropriateness, and correctness. Finally, in the third stage, the inspector needs to check the informal specification to ensure that the user's requirements are formalized completely.

According to the requirement for checking the "necessity" property in the first stage, all of the type declarations, variable declarations, and invariants are checked. Table 6.4 shows the items to be checked and the inspection results. The first two columns in this table give the formal specification items needed to check and their categories. The third and forth columns list the corresponding items that use the items in the first column. The last column indicates the number of conditions of the property given in Definition 12 that describes the relation between the item in the first and third columns. The result of this inspection shows that all the items defined in the formal specification are properly used, and therefore the necessity property is satisfied.

At the second stage of the inspection, the inspector carefully reads the formal specification and answers the questions based on traceability, appropriateness, and correctness. The animation guides the inspector to read the system scenarios defined in the specification. As mentioned in the previous example, more than one system scenario are defined in the target formal specification. Since the inspection of each system functional scenario is

Table 6.5: Operation scenarios of the related processes involved in the system scenario

Process	**Operation Scenario**
Receive_Command	$withdraw\_comm =$ "$withdraw$"$\wedge\ sel = true$
Check_Password	$true\ and\ (exists[x : Account\_file] \mid x.id = id\ and\ x.password = pass\ and\ sel = true\ and\ acc1 = x)$
Withdraw	$amount <= \tilde{}x.balance\ and\ x.balance = \tilde{}x.balance - amount\ and\ cash = amount$

very similar, we only demonstrate the inspection process of one system scenario to avoid the duplication. It is sufficient to allow us to explain all of the major aspects of interest. The inspection of other system scenarios can be understood in the same way.

Suppose the system scenario *{withdraw_ comm} [Receive_ Command, Check_ Password, Withdraw]{cash}* is selected for inspection. The relevant operation scenarios of each process that are given in Table 6.5 are extracted for animation. Since the system scenario contains three operation scenarios derived from the three processes, the inspection of this system scenario is divided into three steps. In each step, the inspector focuses on a specific operation scenario and the related formal specification items. Table 6.6 and 6.7 display the questions for inspecting the selected system scenarios. Since it is impossible to put all of the inspected items and results in the table for the sake of space, we only show one or two cases of inspecting each kind of item in the table and omit the details of other similar cases.

The total 27 questions in Tables 6.6 and 6.7 are asked in the step two of the animation to facilitate the inspection of the second process, *Check_ Password*, of the system scenario. The first 13 questions mainly focus on checking whether all the related formal specification items have corresponding requirements. The next 10 questions are concerned with whether the type declarations, variables, invariants, and processes are defined appropriately and correctly. The last 4 questions help the inspector exam whether the implicit requirements are specified correctly. As a result of this stage, the following errors are found:

1. The invariant "$forall[a, b : Account]|a <> b \Rightarrow a.id <> b.id$" does not realize any constraint in the informal specification.

Table 6.6: The inspection results of the system scenario

NO	**Question**	**Answer**
1	Does operation scenario "*true and* $(exists![x : Account\_file] \mid x.id = id$ *and* $x.password = pass$ *and* $sel = true$ *and* $acc1 = x)$" realize a function?	Yes
2	What is the function?	*F1.1.2*
3	Does this operation scenario realize constraint?	No
4	What variable is used in this operational scenario?	"*x*", "*sel*", "*id*", "*pass*" and "*Account_file*"
5	Whether variable "*x*", "*sel*", "*id*", "*pass*" and "*Account_file*" realize any data store?	Yes
6	Which variable realizes which data resource?	"*Account_file*" realizes *D2.1*
7	What type is used in this scenario?	"*string*", "*bool*", "*Account*", and "*set of Account*"
8	Are type "*string*", "*bool*", "*Account*", and "*set of Account*" defined by designer?	Yes
9	What type is defined by the designer?	"*Account*" and "*set of Account*"
10	Is there any invariant should be applied to this operation scenario?	Yes
11	What is the invariant?	"$forall[a : Account] \mid len(a.id) = 4$"
12	Does invariant "$forall[a : Account] \mid len(a.id) = 4$" realize constraint?	Yes
13	What is the constraint?	*C3.2*
14	Is variable "*Account_file*" defined complying with SOFL?	Yes
15	Does combination of variable "*Account_file*" and type "*set of Account*" appropriately realize the data resource *D2.1*?	Yes
16	Are variable "*x*", "*sel*", "*id*", "*pass*" and "*Account_file*" used appropriately?	Yes

Table 6.7: The inspection results of the system scenario (continue)

NO	**Question**	**Answer**
17	Is invariant "$forall[a : Account] \mid len(a.id) = 4$" defined complying	Yes
	with SOFL?	
18	Does invariant "$forall[a : Account] \mid len(a.id) = 4$" appropriately realize	Yes
	the constraint *C3.2*?	
19	Is invariant "$forall[a : Account] \mid len(a.id) = 4$" used appropriately?	Yes
20	Is the operation scenario specified complying with SOFL?	Yes
21	Can pre-condition "*true*" implies guard condition "($exists![x :$	Yes
	$Account\_file] \mid x.id = id$ *and* $x.password = pass$) *and* $sel = true$"?	
22	Can pre and guard conditions "*true and* ($exists![x : Account\_file] \mid x.id$	Yes
	$= id$ *and* $x.password = pass$) *and* $sel = true$" imply defining condition	
	"$acc1 = x$"?	
23	Does this operation scenario appropriately realize function *F1.1.2*?	Yes
24	Does the function *F1.1.2* use data resource?	Yes, *D2.1*
25	Does this operation scenario use the state variable that realizes the	Yes,
	data resource?	"*Account_file*"
26	Does the function *F1.1.2* comply with constraint?	Yes, *C3.2*, and *C3.3*
27	Does this operation scenario satisfy the invariant that realizes the	Yes
	constraint or does this operation scenario realize the	
	constraint itself?	

2. The constraint "$C3.4$" is not formalized in the formal specification.

3. The implicit requirement that function "$F1.1.4$" complies with constraint "$C3.4$" is not formalized in the formal specification.

4. The implicit requirement that function "$F1.1.4$" uses data resource "$D2.1$" is not formalized in the formal specification.

Taking the same approach, all of the other system scenarios are also inspected at the second stage. As required by our inspection method, the final stage of inspection must devote to the examination of whether the function items in the informal specification are formalized completely in the formal specification. For example, to check the completeness of function "$F1.1$", the inspector forms the conjunction of all the operation scenarios that formalize function items "$F1.1.1$", "$F1.1.2$", "$F1.1.3$", and "$F1.1.4$", and then analyzes whether the conjunction completely formalizes function "$F1.1$". The analysis of such a conjunction can be supported by *Inspection Task Tree* established in [96]. Inspection task tree can decompose a conjunction into a tree structure in which each leaf node presents an atomic predicate. It is a useful technique to help the inspector analyze each part of a conjunction. Since this technique has already made available in the literature, we omit further discussion here for brevity.

6.5 Summary

In this Chapter, we first introduced the four aspects that should be checked in the inspection and formally defined several specific properties for these aspects. Then, we explained how the questions on the checklist are raised based on the four aspects and traceability of the specifications. We also illustrated how to systematically inspect an operation scenario and all related inspection targets. We discussed how to handle the feedback of the inspection and used a case study to demonstrate the entire process of formal specification inspection.

In the next Chapter, we will introduce the supporting tool for our inspection approach. We will explain the architecture of the tool and all the functions it provides. We will also discuss its flexibility in integrating with other functions.

Chapter 7

Tool Support for SOFL Specification Construction and Verification

In this chapter, we present a prototype software tool we have developed to support the entire procedure of our formal specification inspection approach. The tool is actually developed not only for supporting the inspection method, but for providing a framework to support the entire SOFL three-step modeling approach and related techniques.

The framework is implemented in C# programming language under the environment of Microsoft Visual Studio 2008. The current version of the framework supports the construction of SOFL specifications and formal specification inspection. The procedures of inspection that can be supported include system functional scenario generation, animation, checklist construction, and inspection results documentation. Moreover, our framework is designed as a platform to easily integrate other functions. For now, a pattern system for formal specification construction [97] and a prototype parser [98] have been integrated into this framework.

In the rest of this chapter, we first introduce the design and architecture of the support framework. Then, we will explain the functions it provides in detail. We will also discuss how to use these functions to support the inspection.

7.1 Design and Implementation

Since this framework is designed to support the SOFL three-step approach and related techniques, the functions provided by the supporting framework can be roughly separated into two major parts. Figure 7.1 shows the major functions of the framework. The functions in the upper part of Figure 7.1 are used to support the construction

of SOFL specifications, including informal, semiformal, formal, and class specifications. To allow the user to construct the SOFL formal specification, the supporting framework provides both an editor for writing formal module specification and a draw board for drawing CDFD. The CDFD draw board is designed especially for drawing CDFD, and all kinds of components in the CDFD can be easily controlled. Moreover, to facilitate the designer, we provide a convenient function to automatically keep the consistency between the CDFD and the associated module specification.

The functions in the lower part of Figure 7.1 are used to support the SOFL formal specifications inspection. Firstly, the constructed CDFD is used to extract possible system functional scenarios, and the generated system functional scenarios will be used to perform the animation. Then, in the animation process, the information of the informal and formal specifications is used to construct checklist to instruct the inspector carry out inspection. Finally, an inspected formal specification can be delivered to the programmer for implementation.

Except the "Informal Spec. Editor", the other functions shown in Figure 7.1 are implemented by the author of this dissertation himself. More than 40,000 lines of code have been implemented. The details of each function shown in Figure 7.1 is introduced in the next section.

As shown in Figure 7.2, the architecture of the tool can be divided into three layers. The bottom layer is "XML File". It is where the specifications are saved. In our framework, all the specifications, including informal and formal specifications as well as CDFDs, are saved in XML files. The XML files are adopted as bridges to share information between different functions in the "Behavior" layer.

The "Behavior" layer includes two categories of functions for manipulating the XML files. The first category contains the functions that are used to write specifications and draw CDFDs, and the other category includes the functions related to inspection of the formal specification. The user invokes these functions through the interaction with different "*explorers*" in the "Appearance" layer. An "explorer" is a small window in our tool, which displays specific information to the user. To finish different tasks, explorers are organized together into different "Viewers". Each "Viewer" provides necessary information and functions for a specific task.

Using an unified file system to save all the specifications can make functions be implemented independently. All the functions use the specification information saved in the same file format. The developer does not need to consider the implementation of other functions, as long as he implements the function complying with the

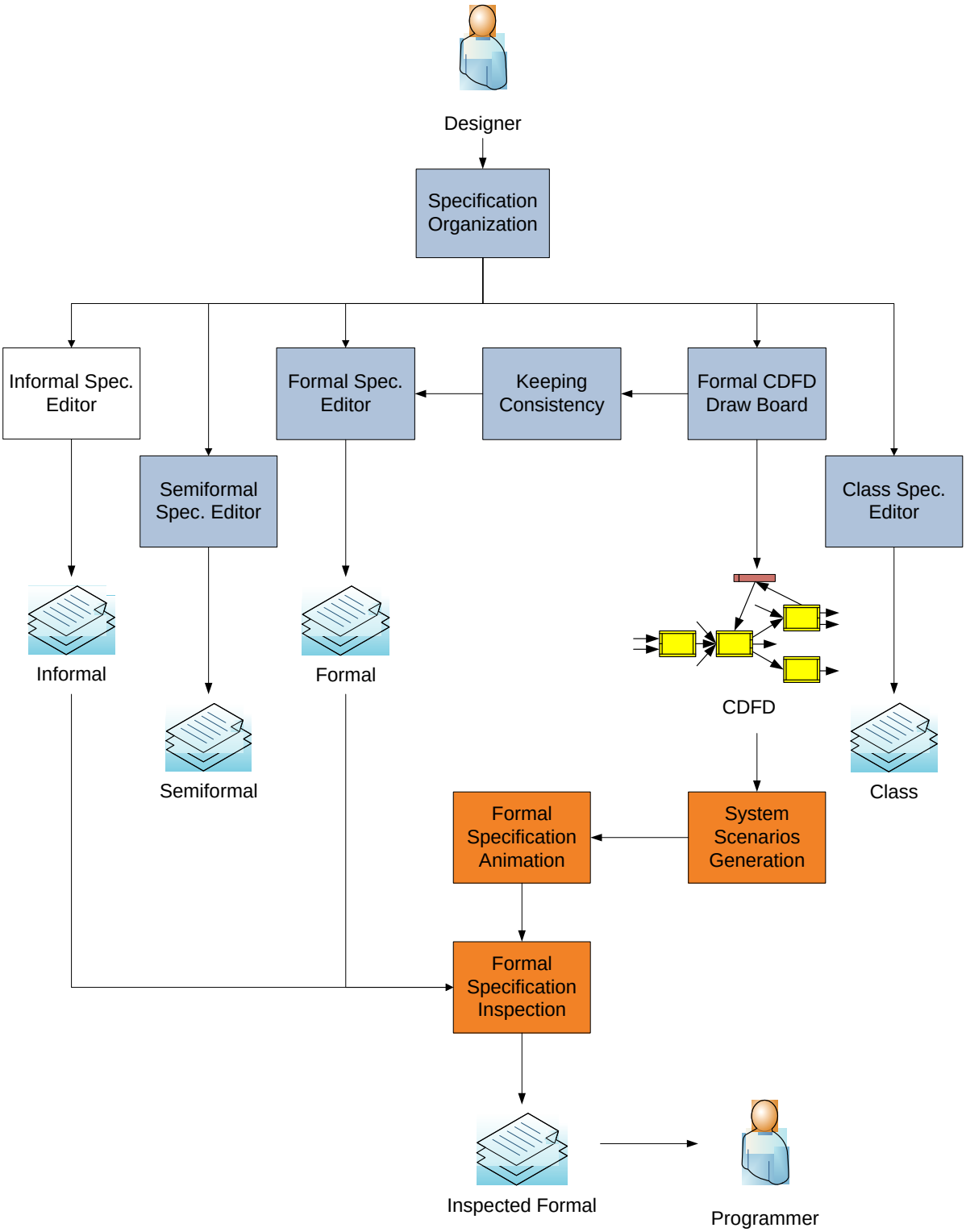


Figure 7.1: The major functions provided by the supporting framework

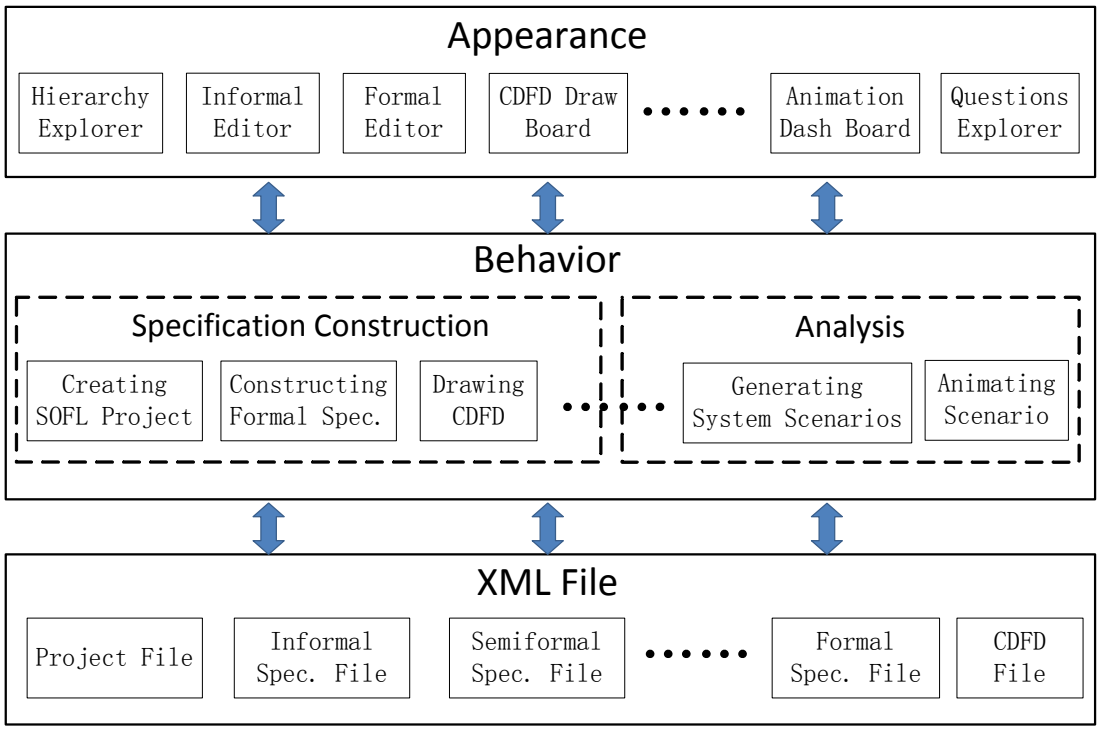


Figure 7.2: The architecture of the framework

pre-defined XML file format. Such a design is also flexible for integrating other functions in the future.

7.2 Functions Provided in the Framework

As mentioned in the previous section, the user finishes a task through a specific "Viewer" in our framework. In this section, we introduce the major functions provided in the framework and corresponding "Viewers".

7.2.1 Specification Organization

According to the development process, a specification for the software system under development is first constructed, and then verified and validated before it is delivered to the programmer for implementation. In the SOFL three-step development approach, four kinds of specifications will be constructed, including informal, semiformal, formal specifications, and class specifications. Each semiformal or formal specification contains a text module specification and an associated CDFD representing the architecture of the system. The class specifications are used to define necessary classes used in the formal specification.

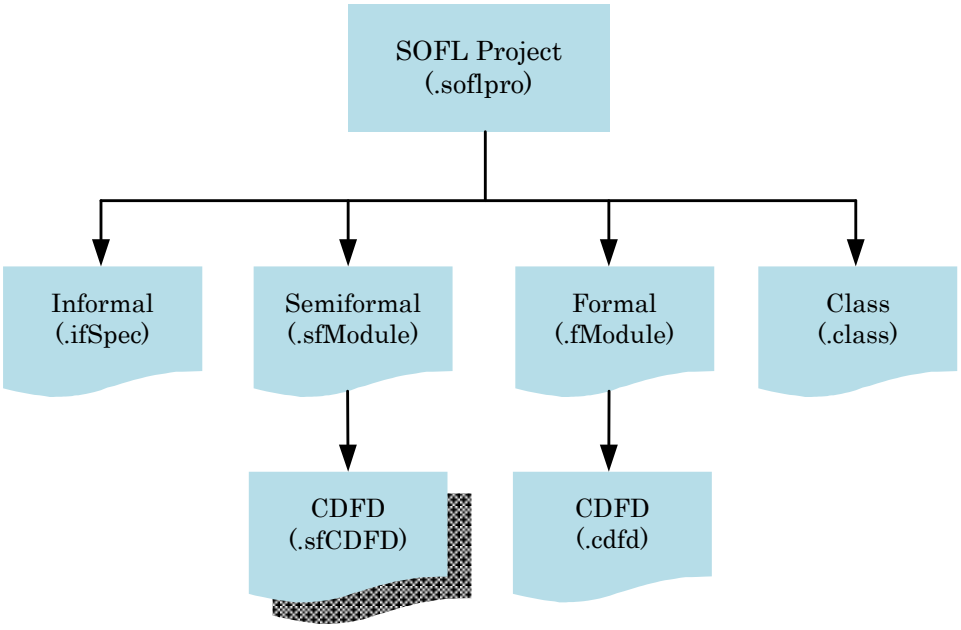


Figure 7.3: The structure of file system used in the framework

In our framework, all the specifications related to the same software system are organized into a SOFL project as shown in Figure 7.3. We use an independent XML file with suffix ".soflpro" to record the information of the XML files used to save specifications.

In the framework, the designer can manipulate the SOFL project through the "Hierarchy Explorer", as shown in Figure 7.4. This explorer is used to present the hierarchy of the specifications contained in a SOFL project. Different kinds of specifications are organized under corresponding tags. User can create a SOFL project and add new specifications to the current project through context menu. When adding a formal module, two tags will be created under the "Formal" tag. The tag with suffix ".fModule" represents the text module specification, and the tag with suffix ".cdfd" represents the corresponding CDFD of the module. The semiformal module has similar structure. The user can also rename or delete a specification. All modifications in the "Hierarchy Explorer" will lead to the update of the XML file used to record the project information.

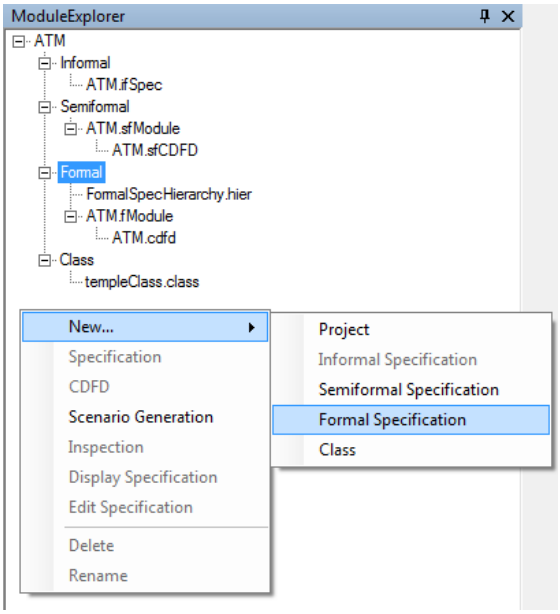


Figure 7.4: The "Hierarchy Explorer" used to manipulate the SOFL project

7.2.2 Informal Specification Editor

The first step of the development process is the construction of an informal specification. In our framework, only one informal specification is written in a SOFL project. The corresponding XML file will be created in the creation of a SOFL project. The explorer "Informal Editor" is used to edit the informal specification. It can be opened by double clicking the corresponding tag in the "Hierarchy Explorer".

Figure 7.5 shows the "Viewer" for specifying informal specification. The explorer on the left-hand side is the "Hierarchy Explorer" introduced in the previous section, and the explorer on the right-hand side is the "Informal Editor". The structure of the informal specification is fixed in the "Informal Editor". The three key words "Functions", "Data Resources", and "Constraints" are automatically created for the informal specification. The number of each requirement item is also automatically maintained. By pressing the "enter" key, the user can create a new line to specify a new requirement item. The sequence number of this new item will be created automatically and follows the previous item. The "tab" key and "shift + tab" combined key can move a requirement item to a lower level and higher level, respectively.

For example, if the user press the "enter" key at the end of the requirement item "F1.1", the editor will create a

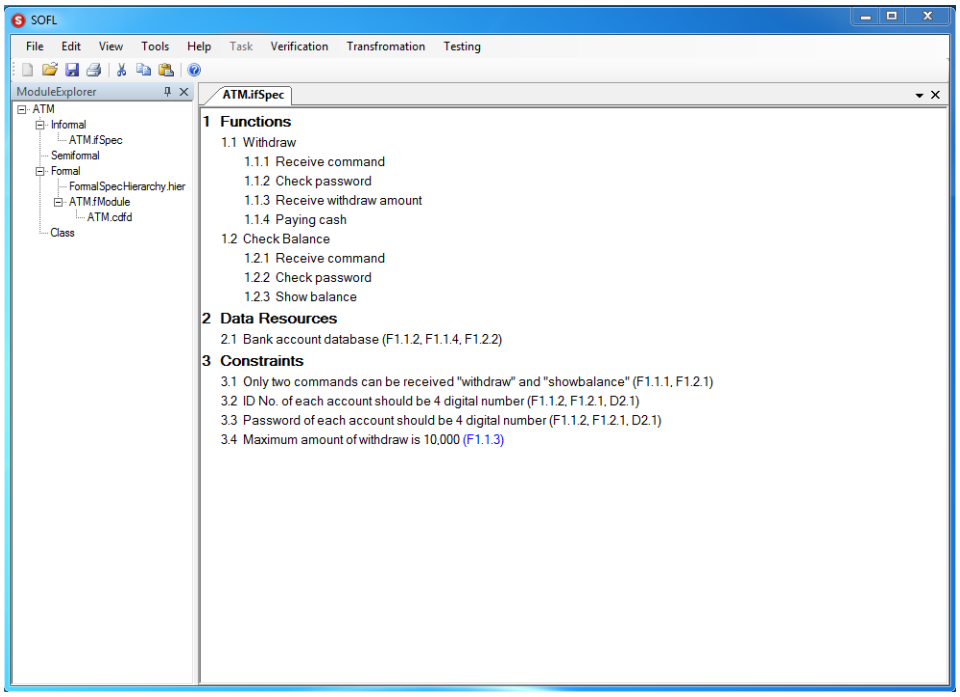


Figure 7.5: The user interface for constructing informal specification

new line starting with "F1.2". The user can specify a new function in this new line. If the user wants to decompose the function "F1.1", he can press "tab" key to move the new item to lower level and the sequence number will become "F1.1.1" automatically. The informal specification is saved in the XML file using the same structure.

7.2.3 Semiformal and Formal Specifications Editor

The semiformal and formal specifications can be constructed on the basis of the informal specification. Since the structures of these two kinds of specifications are very similar, our framework provides similar functions to support the construction of both kinds of specifications. Here we only introduce the user interface and related functions for constructing formal specification to avoid duplication.

As mentioned before, a formal specification in SOFL includes a CDFD and the associated module. In principle, the CDFD is first drawn and the associated module is then completed. Figure 7.6 shows the snapshot of the interface for editing a formal specification. The center of this interface is an explorer called "CDFD Draw Board". The tool bar on the top of this explorer lists the components designed specifically for SOFL CDFD, user can add

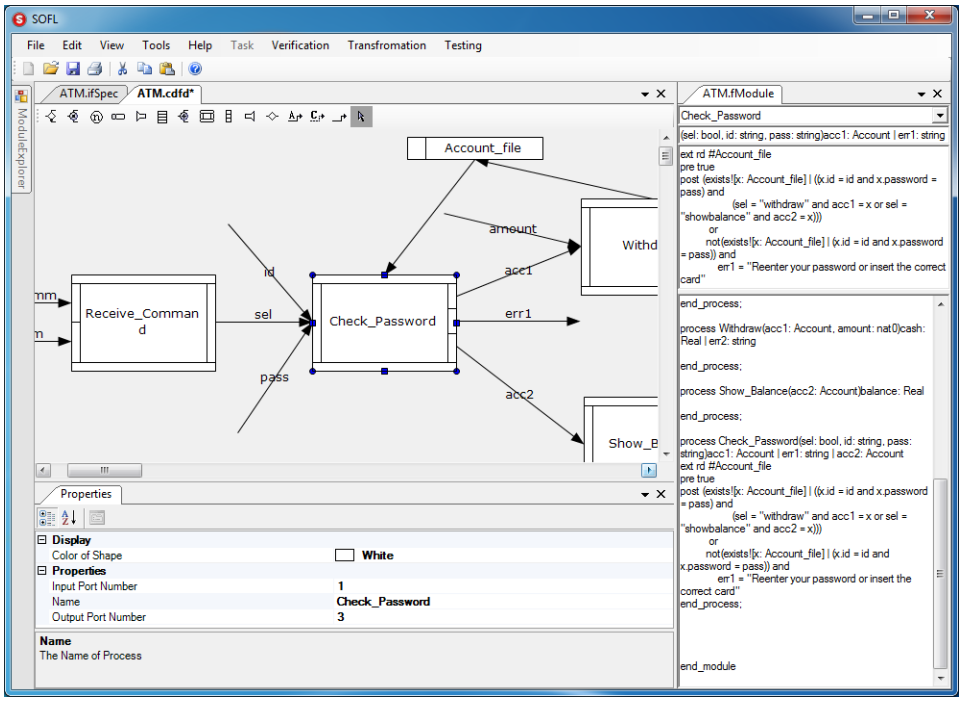


Figure 7.6: The user interface for constructing formal specification

a component to the CDFD by clicking corresponding icon.

The attributes of each component can be modified in the "Property Explorer", which is at the bottom of the snapshot shown in Figure 7.6. If the user wants to change some attributes of a component in the CDFD, he first needs to select the component in the "CDFD Draw Board", and all attributes of this component will be listed up in the "Property Explorer". Any modification made in the "Property Explorer" will be reflected in the "CDFD Draw Board" directly.

For example, a process named "Check_Password" is selected in Figure 7.6. The attributes of the process "Check _Password" are listed in the "Property Explorer", which include "Name", "Input Port Number", "Output Port Number", and "Color of Shape". As mentioned in Section 2, in the CDFD, the small rectangles on the left and right sides of a process represent its input and output ports respectively. The "Input Port Number" and "Output Port Number" indicate the number of input and output ports. As shown in the "Property Explorer", the value of "Input Port Number" is 1 and the value of "Output Port Number" is 3. Correspondingly, the process in the CDFD contains one small rectangle on the left side and three small rectangles on the right side. If the user changes

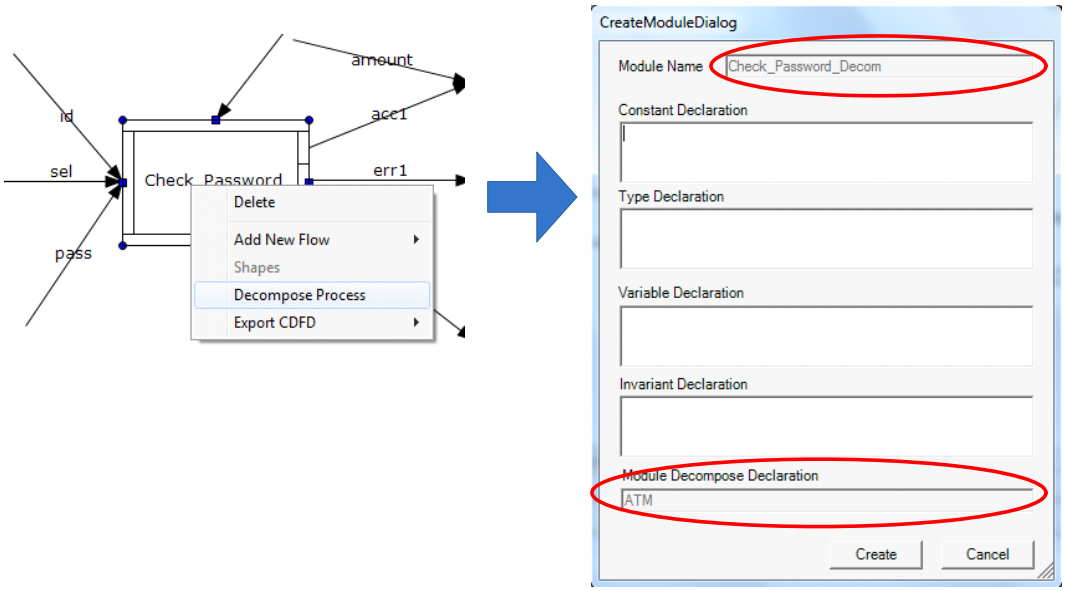


Figure 7.7: Creating new module by decomposing a process

the value of "Input Port Number" or "Output Port Number", the number of the small rectangles will be changed automatically. The attribute "Name" is the text displayed in the center of the rectangle that represents the process, and the user can change the color of the rectangle by changing the attribute "Color of Shape".

Moreover, in the "CDFD Draw Board" we provide an alternative way to create a new module. The user can decompose a process in the CDFD directly to create a lower level module. As shown in Figure 7.7, by clicking the "Decompose Process" item in the context menu, the dialog used to create new module will be popped up. The highlighted parts of this dialog is automatically filled, and it indicates that the new module is decomposed from process "Check_Password" in module "ATM". Using this approach to create new module is more straightforward and complying with the essential idea of SOFL formal specifications.

The other explorer shown in Figure 7.6 is the "Formal Editor" that is used to specify the text module specification. In this explorer, the user is not allowed to edit the entire formal specification directly. Instead, the user edits a specific part of the specification each time. On the top of this explorer, there is a drop-down list that includes all possible parts of a module specification that can be edited. As shown in Figure 7.8, the formal specification is divided into several declaration parts and processes. The declaration parts include the constant identifier, type, and other declarations introduced in Chapter 2. The user can select a specific declaration part or a process to edit.

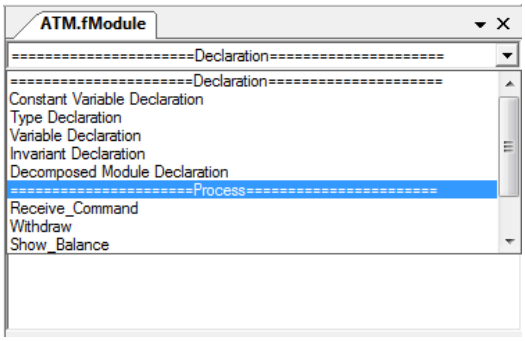


Figure 7.8: The drop-down list in the "Formal Editor"

For instance, in the "Formal Editor" in Figure 7.6, the process "Check_Password" is selected for editing. The text box under the drop-down list presents the input and output variable lists of the process "Check_Password". The user can edit the contents of the process in the rich text box under the variable lists. At the bottom of the explorer, the entire formal specification is displayed. Any change to the content of the specification will be reflected here immediately.

We design the "Formal Editor" like this rather than a plain text editor since it can facilitate us to provide the function that always keeps the consistency between the CDFD and the module. In our framework, whenever the CDFD is changed, the module will also be properly updated automatically. Separating a formal specification into different editable parts can make this process easier. The formal specification contents that correspond to the change of the CDFD can be easily located without parsing the specification.

7.2.4 Keeping Consistency between the CDFD and the Module

In a SOFL formal specification, the CDFD is the graphical presentation of a module: it shows how different processes are connected and the signature of each process. The signature of a process is similar to the concept in the programming language, it includes the name of the process, the input and output ports number, and the input and output variable lists. In the CDFD, a process is represented as a rectangle and its name is displayed in the center of the rectangle. The small rectangles on the two sides of the process represent the input and output ports, and the dataflows that connect to each port indicate the input or output variables.

In the construction of SOFL formal specification, drawing CDFD and specifying module are two actions. If

the specification needs to be changed, the designer should change both the CDFD and the module. Keeping the consistency between the CDFD and the module will cost the designer a lot of time and effort, especially when the formal specification is frequently changed. In order to facilitate the designer to construct the formal specification, our framework provides a significant function to automatically keep the consistency.

In the framework, we use the following strategy to keep the consistency between the CDFD and the module:

1. If a content in the formal specification can be changed in the CDFD, it should be changed in the CDFD. That means the change of the contents that are shared by the CDFD and the module should be made in the CDFD.

2. The same content in the formal specification can only be changed in one place. It means the same information can be changed either in the CDFD or in the module. If a content is changed in the CDFD, the change should be automatically reflected in the module.

Using this strategy to keep the consistency is reasonable. In principle, the CDFD should be drawn before the associated module is specified. Therefore, if some contents in the formal specification need to be changed, they should be changed in the CDFD. It is consistent to the procedure of constructing a formal specification. Furthermore, asking the designer to change a content in one place and using the program to change the content in the other place can avoid the mistakes made by human.

As mentioned earlier, the intersection between the CDFD and the module is the signature of processes. Based on our strategy, the signature of the processes should be modified in the "CDFD Draw Board" through the "Property Explorer", and any modification will be automatically reflected in the "Formal Editor". For example, if the designer wants to change the name of process "Check_Password", he should first selected the process in the "CDFD Draw Board", and then change the value of attribute "Name" in the "Property Explorer". The new name of the process will be used to replace the old name of the process in the "Formal Editor".

A data flow in the CDFD represents a variable in the module. Each data flow has two attributes "Name" and "Type", and its name is displayed in the CDFD. The designer can change these two attributes in the "Property Explorer" like other CDFD components. A data flow in the "CDFD Draw Board" can connect to a process by docking to one of its port. By docking, we mean the data flow is "stuck" on the port of the process. When the

Table 7.1: The changes that may affect the consistency

NO	**Event**	**Responding Action**
1	Change process's name	Change the process's name in the module
2	Change process's input port number	Disconnect the data flows connecting to the input port of this process, and erase the input variables list in the module
3	Change process's output port number	Disconnect the data flows connecting to the output port of this process, and erase the output variables list in the module
4	Add a new process	Add a new process to the module
5	Delete a process	Disconnect all the data flows connecting to this process, and delete the process from the module
6	Change data flow's name	If the data flow connects to processes, change the variable's name in the variable list of the processes in the module
7	Change data flow's type	If the data flow connect to processes, change the variable's name in the variable lists of the processes in the module
8	Add a new data flow	If the data flow connects to processes, add the new variable to the variable list of the processes
9	Delete a data flow	If the data flow connects to processes, delete the variable from the variable list of the processes
10	Connect a data flow to a process	Add the new variable to the variable list of this process
11	Disconnect a data flow from a process	Delete the variable from the variable list of this process

end of a data flow is close to a port in a certain range, this data flow will be automatically "stuck" to the port. If the designer moves the process, the data flow will be moved accordingly. A data flow that does not connect to any process in the CDFD is not reflected in the associate module. The corresponding variable of a data flow will not be listed in the variable list of a process until the data flow connects to the process.

Table 7.1 lists up the events that may lead to the modification of the associate module. The second column is the modification made to the CDFD, and the third column describes what will be done by our framework to keep the consistency. For example, the modification in the sixth line is to change the name of a data flow. The program will first check whether the data flow connects to any process. If the data flow does not connect to any process, the modification does not affect the module since the data flow is not reflected in the module. If the data flow does connect to a process, the program needs determine which port of the process it connects to. Finally, the program will update the variable list of the process in the module, and all the modification will be reflect in the "Formal Editor" automatically.

7.2.5 System Functional Scenarios Generation

Generating possible system functional scenarios from a formal specification is a major step to perform our inspection approach. The algorithm explained in Chapter 4 has been implemented in our framework. After a CDFD is drawn and all the relevant processes are defined in the associated module, all possible system functional scenarios can be derived based on the topology of the CDFD automatically. The user can use the context menu in the "Hierarchy Explorer" to derive system scenarios from the selected module or CDFD.

Figure 7.9 shows the snapshot of generating system scenarios. The explorer in the center of Figure 7.9 is "CDFD Displayer". Unlike the "CDFD Draw Board", the user is not allowed to manipulate the CDFD in this explorer. Since the system scenario generation function is an independent function, any change of the CDFD in the "CDFD Draw Board" will not be reflected in the "CDFD Displayer".

The explorer "Scenario Explorer" at the bottom of the snapshot lists all possible system scenarios derived from the CDFD. Although some derived scenarios may not be meaningful in representing desired behaviors due to the fact that the derivation is done merely based on the topology of CDFD without analyzing the semantics of the processes involved, they can be easily detected during an inspection.

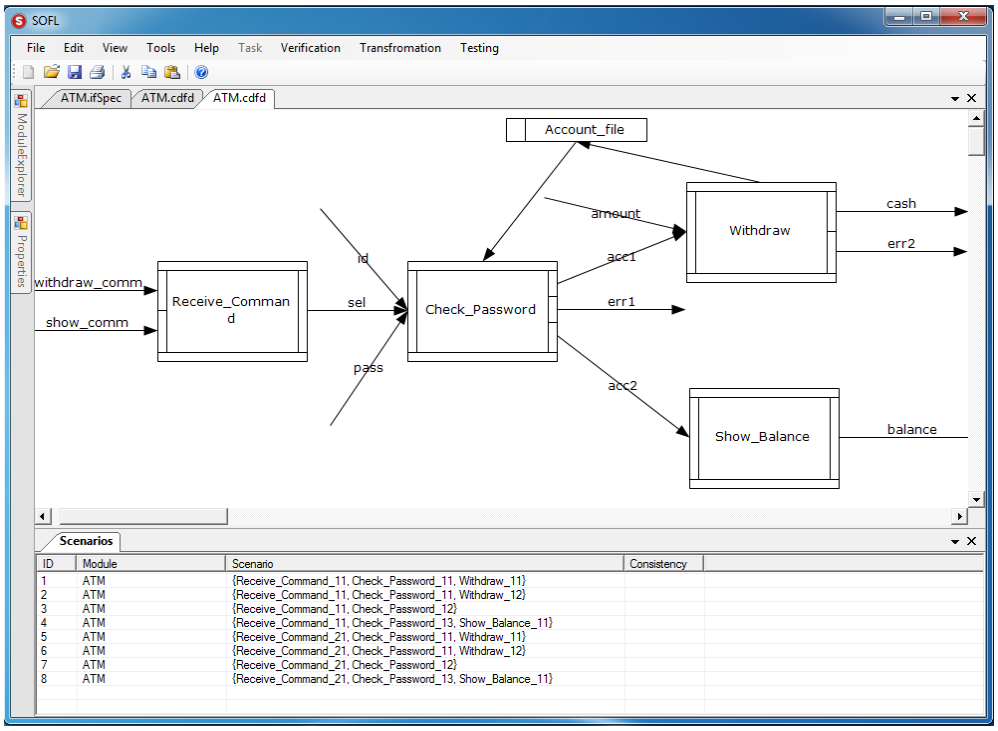


Figure 7.9: The snapshot of generating system scenarios

7.2.6 Animation

After all possible system scenarios are generated, the inspector can select one of the system scenarios for inspection. For example, if the scenario {*withdraw_ comm*} [*Receive_ Command*, *Check_ Password*, *Withdraw*] {*cash*} is selected, the inspector can click the "Animation" item in the context menu to animate the scenario. Figure 7.10 shows the interface for animating a specific system scenario.

The structure of the snapshot in Figure 7.10 is similar to the snapshot shown in Figure 7.9, the only difference is that at the bottom of the snapshot the explorer "Animation Dash Board" overlaps the explorer "Scenario Explorer". This new explorer provides a spreadsheet for displaying all of the related variables and their types automatically derived from the specification, and corresponding values of the variables are inputted by the user of the tool. These values are actually the test suites introduced in Chapter 4, and they will be displayed in the CDFD in the animation.

At the top of the explorer "Animation Dash Board" is a tool bar that contains several buttons for controlling the

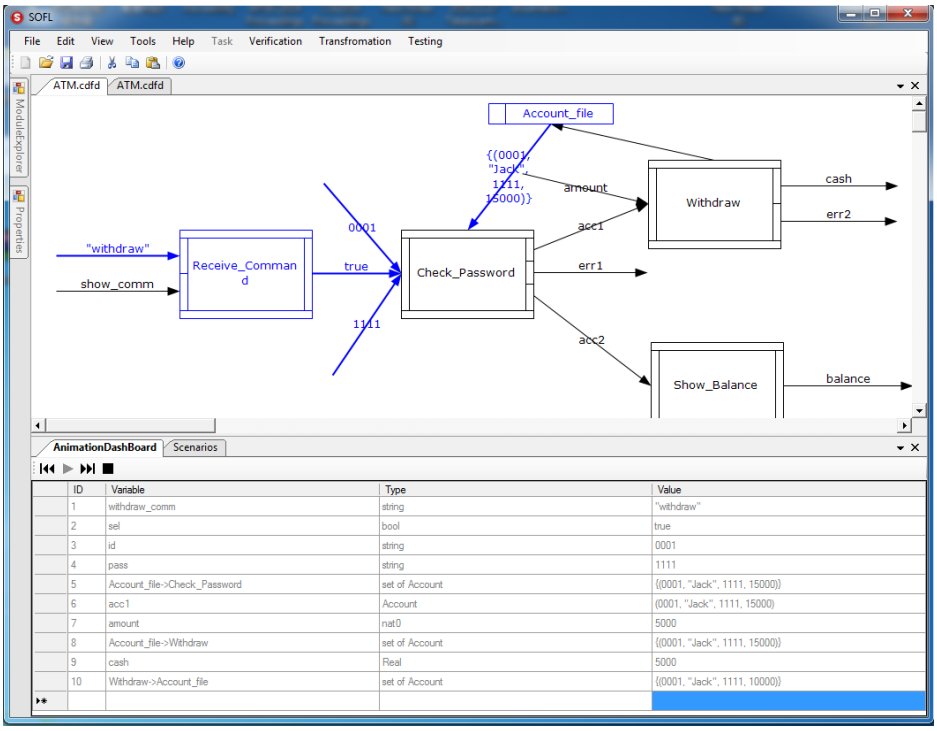


Figure 7.10: The snapshot of formal specification animation

animation process. The buttons allow the user to control the animation step by step or play the entire animation continuously. In each step of the animation, the input data flow, output data flow, and the graphical representation of the process involved in the animation will be highlighted. The test suites will replace the names of corresponding data flows on the CDFD.

7.2.7 Inspection

In our inspection approach, the inspector checks each process involved by following the animation procedure. Figure 7.11 shows the interface for inspection. To inspect a system scenario, the inspector control the animation step by step. In each step, a process is animated, then, an explorer known as "Question Explorer" will be popped up. The "Question Explorer" displays the questions mentioned in Chapter 6 to facilitate the inspection of the operation scenario and the related inspection targets.

The questions are asked in an interactive manner in the "Question Explorer". That means only one question is displayed each time, and the inspector can press the "Next" button to proceed to the next question. As shown

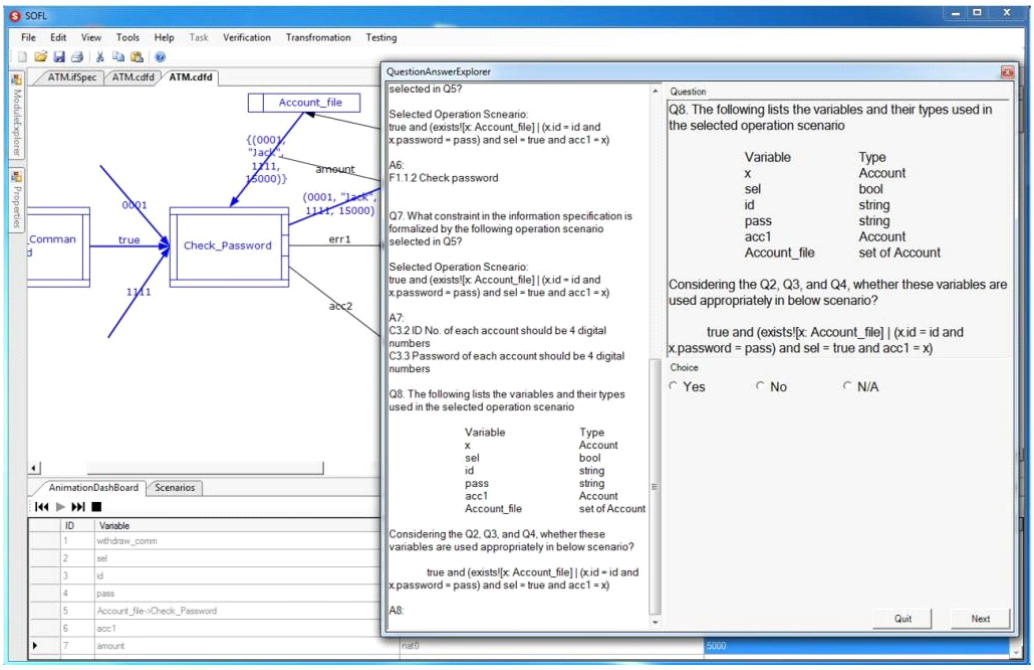


Figure 7.11: The snapshot of formal specification inspection

in Figure 7.11, a specific question is displayed on the right side of the "Question Explorer", and all the questions and their answers are displayed on the left side of the explorer.

We make such design for two reasons. One reason is that the interactive manner can help the inspector to focus on the current question. If all the questions are lists together, the inspector may have chance to miss some important questions by carelessness. The other reason is that the later questions on the checklist may need the answers of the previous questions. Using the interactive manner can ensure the necessary answers are available to the later questions. After answering all the questions on the checklist, the inspection for one operation scenario is completed. The animation will proceed to the next operation scenario occurring in the current system functional scenario and the same inspection procedure will be repeated.

Note that, our framework uses a built-in question template to automatically generate specific questions for each inspection target. The specific question can remind the inspection what exactly should be checked. However, the specific checklist cannot be created if the related inspection targets cannot be extracted from the formal specification. Since the lack of a powerful parser for the formal specification, the current version of our framework asks the user to input the inspection targets for each system scenario.

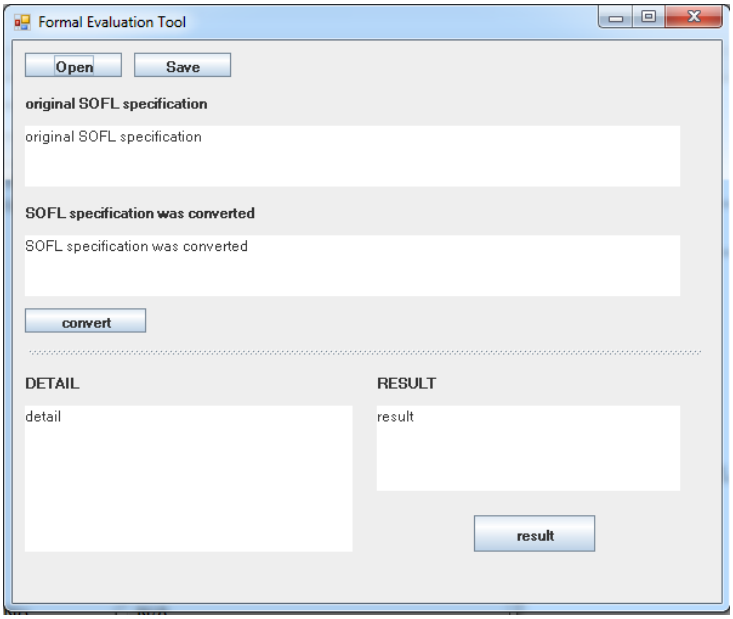


Figure 7.12: The "evaluator" for operation scenarios

7.2.8 Integrated Functions

The framework introduced in this chapter is not built for the formal specification inspection exclusively. It is designed for the entire SOFL develop process and related techniques. The well defined file system and interface make the framework flexible and easy to integrate other functions. In addition to the built-in functions, the current version of the framework also integrates the functions developed by other researchers.

The "evaluator" provided in the framework is used to evaluate the operation scenarios based on test suites. The interface of the "evaluator" is shown in Figure 7.12. It is implemented in Java programming language by a master student in our research team and has been integrated in our framework. In the "evaluator", the operation scenario is first transformed into a Reverse Polish expression, then the test suite will be used to replace the corresponding variables for evaluating the expression.

Another integrated function is an pattern system [97] that is used to help the designer easily construct the formal specification. As shown in Figure 7.13, the user can call this pattern system by using the context menu in the editable area of the "Formal Editor". The pattern system will insert the constructed formal specification segment to the place where the cursor is.

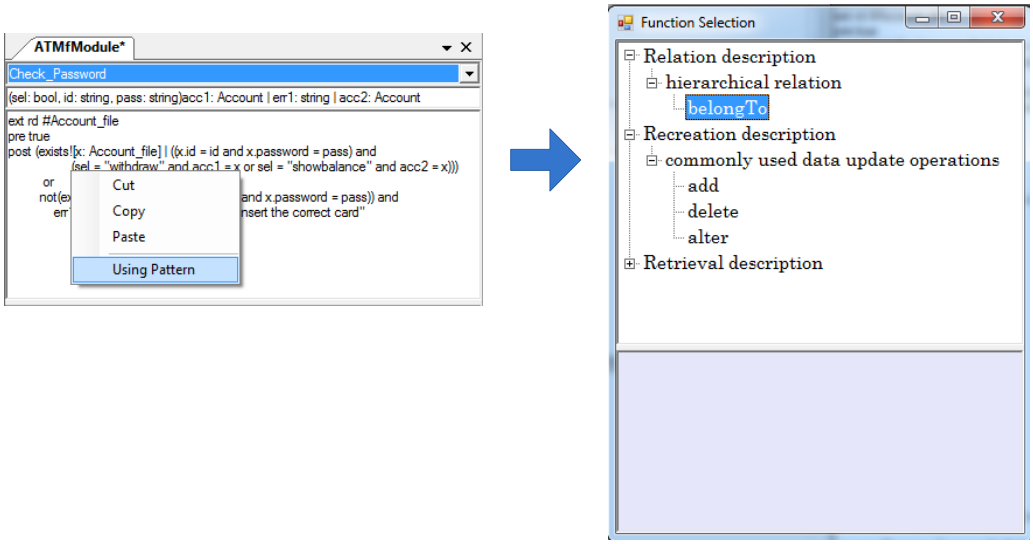


Figure 7.13: Invoking the integrated pattern system function

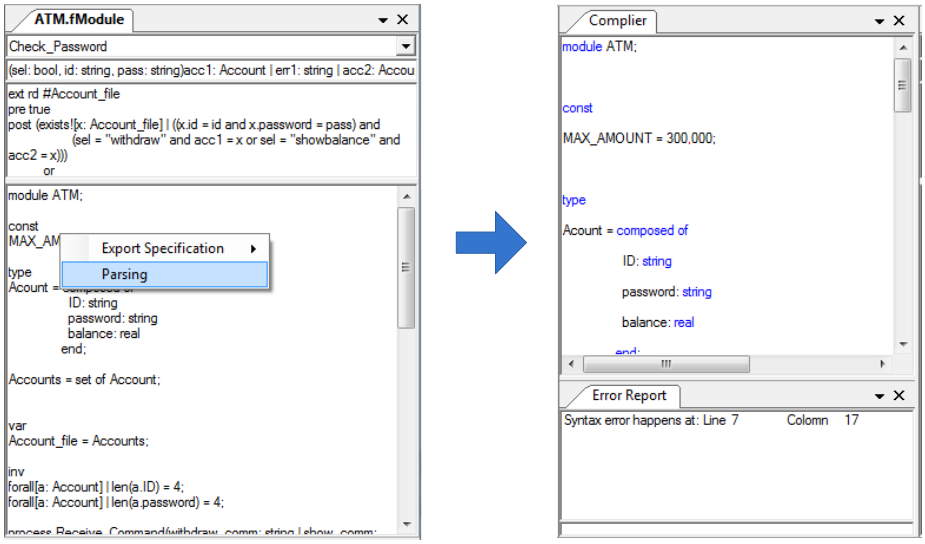


Figure 7.14: Invoking the integrated parser function

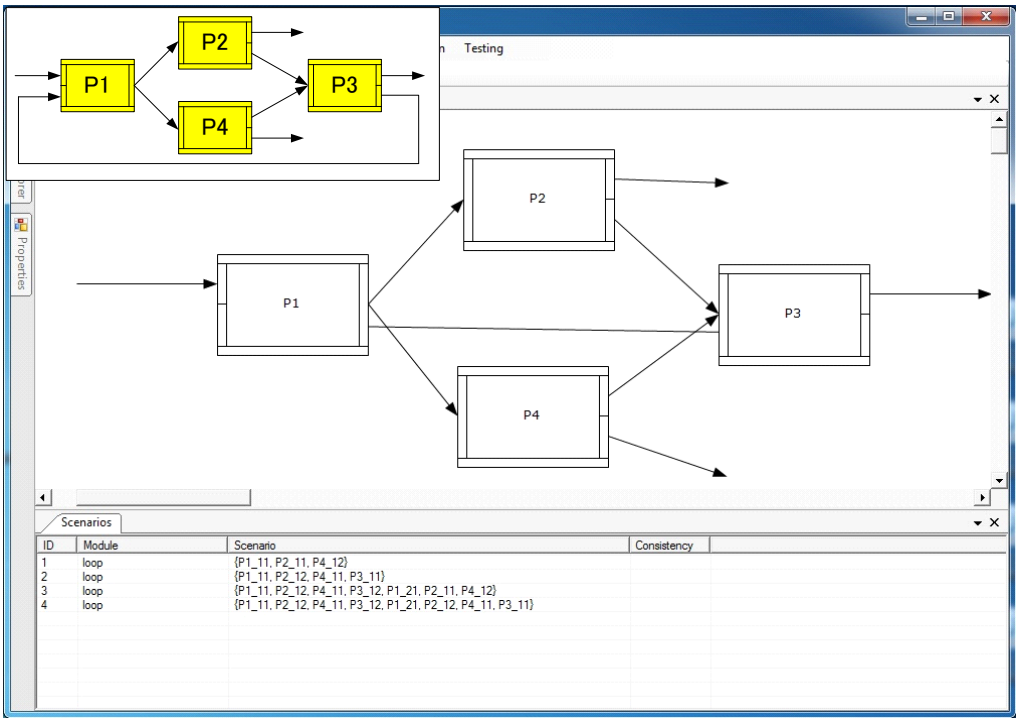


Figure 7.15: System scenario generation from a CDFD with single loop

The other integrated function is a lightweight parser [98] for the formal specification, which can only do simple syntax checking. The user can call the parser by using the context menu in the display area of "Formal Editor". Figure 7.14 shows the explorer used to display the parsing result.

7.3 Evaluation of System Scenario Generation Function

As discussed in Chapter 4.2.5, the number of combinations between process input ports and output ports will increase rapidly with the increasing complexity of the CDFD. In this section, we first use two examples to demonstrate the loop structures that can be handled by our supporting tool, then, we use a more complex CDFD to evaluate the combinatorial explosion problem.

7.3.1 Loop Structures

The CDFDs used in the two demonstrations are the CDFDs shown in Figure 4.5 and 4.9 in Chapter 4. Figure 7.15 shows the system scenarios generated from the CDFD shown in Figure 4.5. The upper left part of Figure

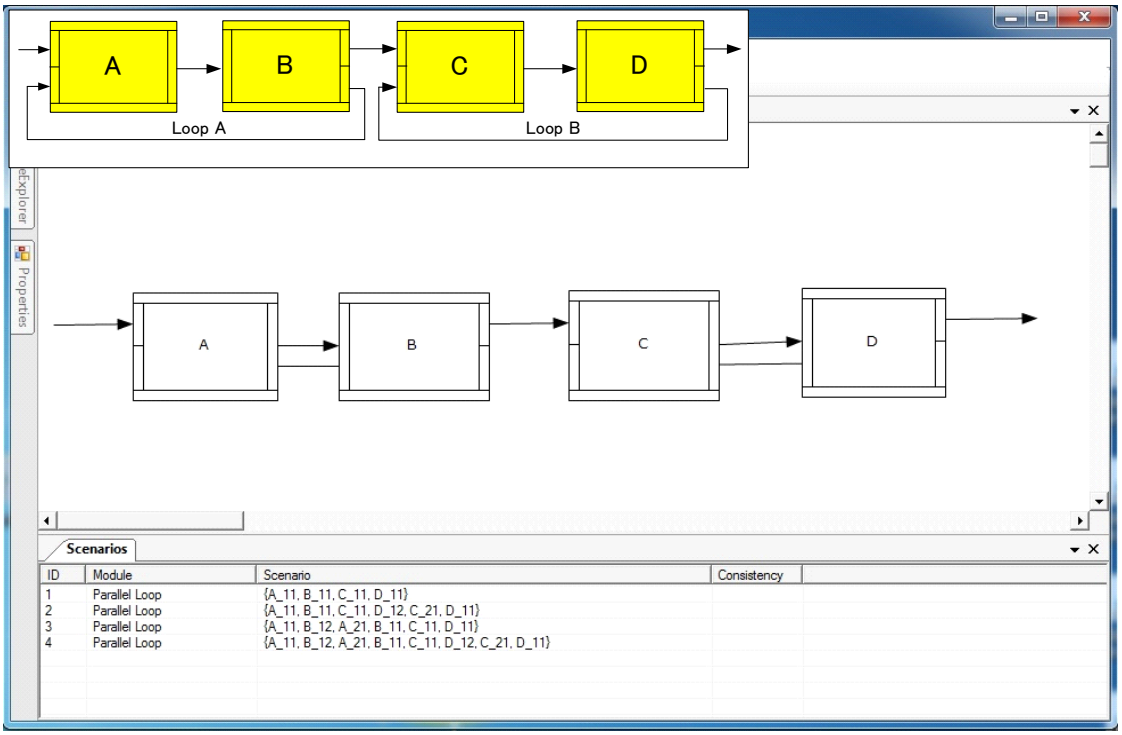


Figure 7.16: System scenario generation from a CDFD with sequential loops

7.15 presents the original CDFD. Since the current version of the supporting tool provides only straight line for drawing CDFDs, the data flow that cases the loop cannot be seen clearly in Figure 7.15. Some part of this data flow is coved by process “P1” and “P3” and therefore only part of the data flow can be seen at the center of the CDFD.

By using the system scenario generation function of the supporting tool, four system sceanrios are derived from the CDFD, as explained in Chapter 4.

The second example is deriving system scenarios from the CDFD shown in Figure 4.9. In this CDFD, two loops are connected sequentially. Figure 7.16 shows the generation results. As mentioned in Chapter 4.2.4, four system scenarios are generated from the CDFD.

7.3.2 Evaluation of Combinatorial Explosion

In order to evaluate the combinatorial explosion scale of the CDFD with complex structure, we use the supporting tool to generate system scenarios from a CDFD containing 30 processes. Each process in the CDFD has three

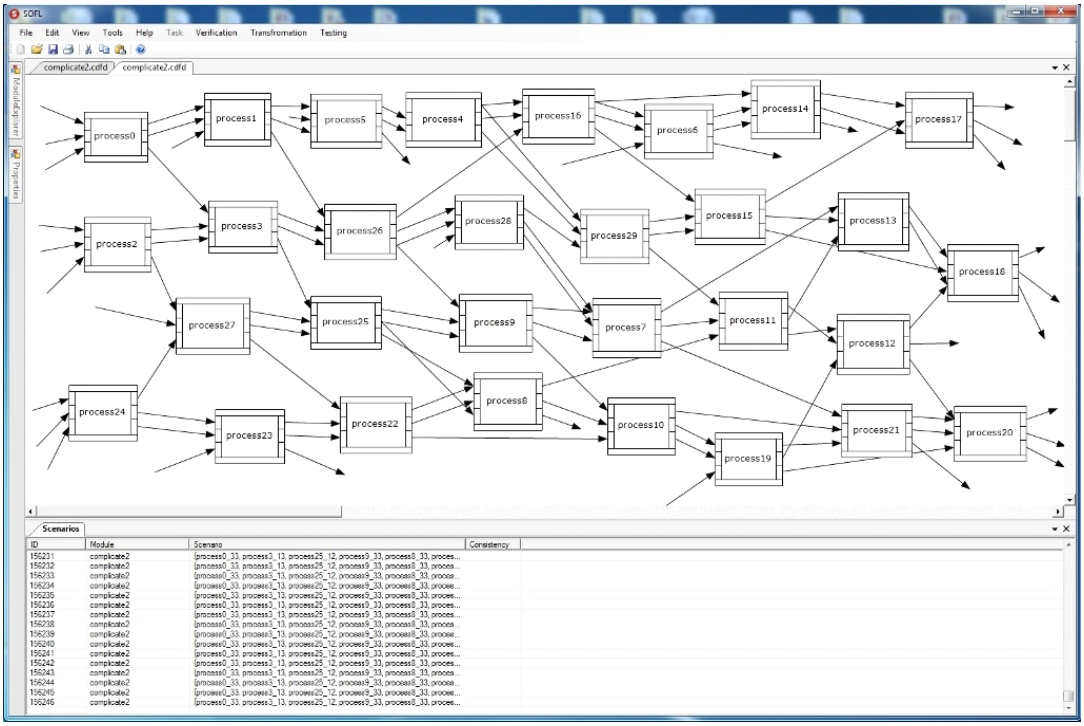


Figure 7.17: System scenario generation from a CDFD with 30 processes and no loop structure

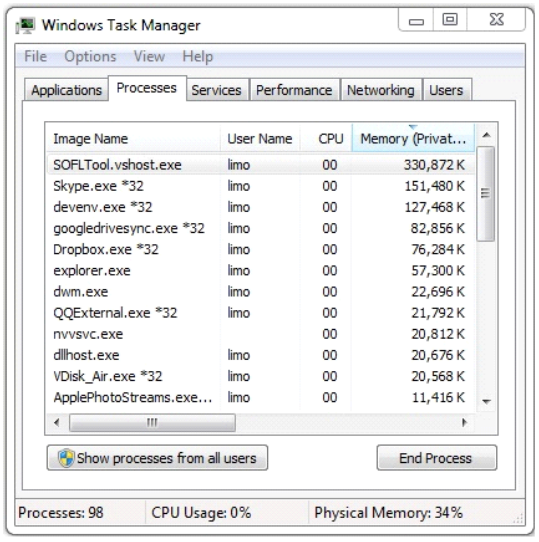


Figure 7.18: The main memory usage of generating system scenarios from a CDFD with 30 processes and no loop structure

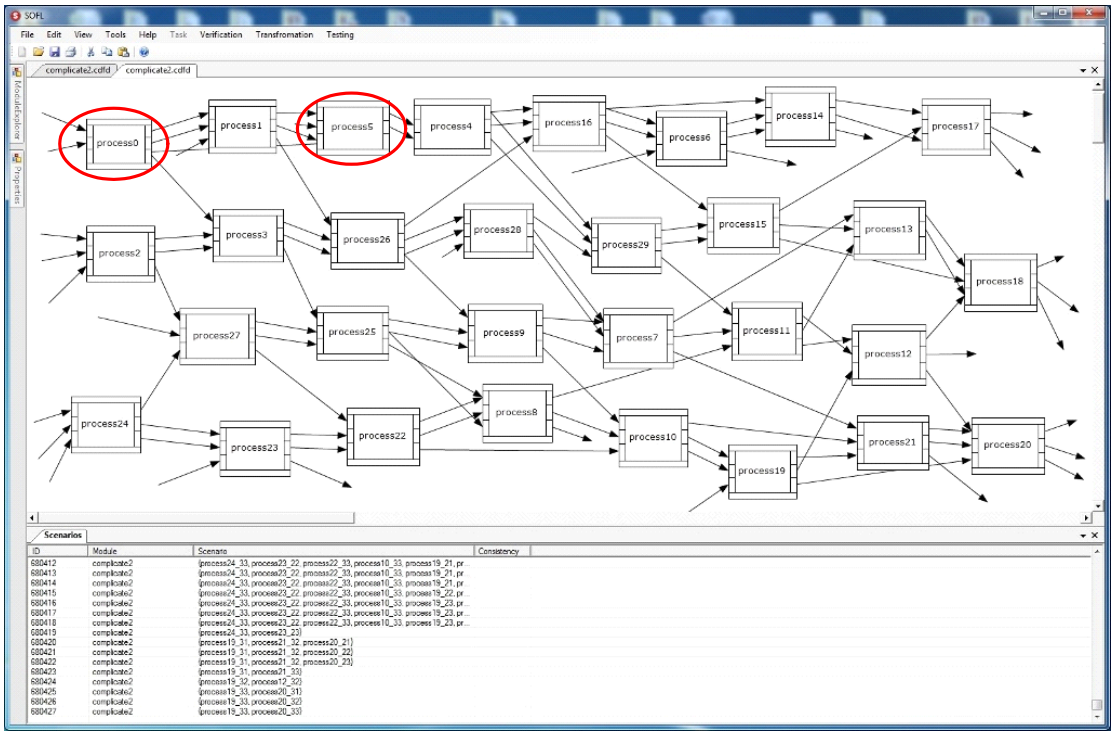


Figure 7.19: System scenario generation from a CDFD with 30 processes and one loop structure

input ports and three output ports. We first generate system scenarios form the CDFD with only sequence and parallel structures, as shown in Figure 7.17. The supporting tool uses around 1 minute to generate 156246 system scenarios. The usage of main memory is shown in Figure 7.18. We use the "*Windows Task Manage*" provided by Windows 7 Operating System to monitor the usage of the main memory. As shown in Figure 7.18, the supporting tool uses 330,872K of memory to generate 156,246 system scenarios from the CDFD.

Since the loop structure may increase the complexity of a CDFD, we also use the supporting tool to test the case of generating system scenarios from a CDFD with loop structure. As shown in Figure 7.19, the CDFD contains one loop from the third output port node of process "*process5*" to the third input port of process "*process0*", which are highlighted. The other part of the CDFD is the same as the CDFD in Figure 7.17. The supporting tool uses around 3 minutes to generate 680,427 system scenarios from the CDFD, which is 435% of the number in the previous CDFD. This means the one loop contained in the CDFD increases the number of system scenarios by 4.35 times. By monitoring the usage of memory, we found that the supporting tool uses 1,840,344K of memory to generate the system scenarios, which is the 556% of the memory cost in the previous example. Moreover, there is

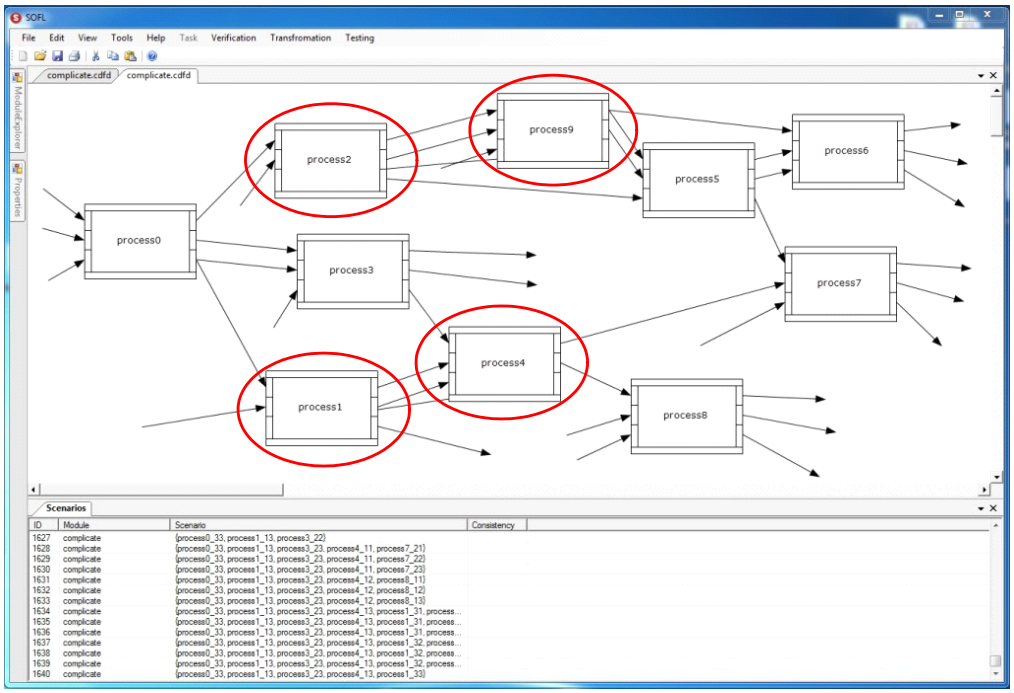


Figure 7.20: System scenario generation from a CDFD with 10 processes and two loop structures

no inspector willing to inspect the entire 680,427 system scenarios.

In addition to the previous extreme cases, we also use the supporting tool to generate system scenarios from a CDFD with normal scale. As shown in Figure 7.20, the CDFD contains 10 processes and each process has three input ports and three output ports. Two loop structures exist in the CDFD. One is from the third output port of process "*process9*" to the third input port of process "*process2*", and the other is from the third output port of process "*process4*" to the third input port of process "*process1*". The processes involved in the loop structures are highlighted in Figure 7.20. The supporting tool uses 46,944k of memory to generate 1,640 system scenarios from the CDFD.

As indicated by the above examples, the CDFDs with complex structures contain huge amounts of possible system scenarios and the supporting tool costs too much memory to generate system scenarios from a complex CDFD. Although the last example indicates that the performance of the supporting tool is good enough to handle the most common cases in our normal research, we plan to improve both the algorithm and the implementation of the system scenario generation function to make it more efficient.

7.4 Experience of Using the Tool

The functionality of our framework can be separated into two parts. The first part is to help designer to construct SOFL specifications. This part of the framework has been used in the teaching of SOFL for at least two years. Over one hundred students have used the functions of the informal and formal specification editors to build specifications for many applications. This practice allows us to have improved the quality of the tool significantly in both functions and the GUI structure.

The other part of the framework includes the functions to manipulate and analyze the formal specification, namely the scenario generation function, the animation function, and the inspection function. The scenario generation function and animation function have been widely used in public presentations to demonstrate SOFL approach and related concepts. Since the inspection targets of the formal specification cannot be extracted automatically at present and have to be inputted by the user, the inspection function has not been widely used. However, the function has been well tested for evaluating its usability and stability. The experience shows that the framework is of high quality, stable, and efficient in supporting both specification construction and specification animation-based inspection.

7.5 Summary

In this Chapter, we introduced a framework for supporting the SOFL specification construction and the entire specification inspection procedure. This framework is also an platform to integrate third-party functions. The three-layer structure and the well designed interface make the framework flexible for further extension. We introduced the functions provided by the framework, and we also indicated how to use these functions to support the formal specification inspection.

Chapter 8

Experiment

In this chapter, we present an experiment on the performance of our formal specification inspection method. Since the inspection is mainly conducted by human, we invited 55 subjects to join the experiment to avoid the bias made by individual inspector. The subjects are divided into three groups to inspect the same formal specification by using two different inspection methods.

In order to create an experimental environment similar to the practice, we choose students as the subjects of our experiment. This is because the students have experience in both requirement formalization and programming, and they are similar to practitioners in industry.

In this experiment, we intend to evaluate our inspection approach from the following aspects:

1. Whether our inspection approach can perform better than other inspection method if the experience of the inspectors are at the same level.

2. Whether the facility provided by our inspection approach can offset the difference on the experience of different inspectors.

3. Whether the domain knowledge of inspector affects the performance of our inspection approach.

4. Whether the traceability related questions adopted in our inspection approach can help inspector detect more defects that affect the consistency between the informal and formal specifications than other inspection method.

In the following of this Chapter, we first introduce the experimental environment, then we explain how the experiment is conducted, and finally we analyze the results of the experiment and point out the findings we found in this experiment.

8.1 Experiment Settings

In this section, we introduce the environment of our experiment. The background of the target formal specification is first explained briefly. Then, we discuss how the subjects are selected and the approach we used to conduct the experiment.

8.1.1 Background of the Target Specifications

The target formal specification in our experiment describes a security transaction system. Similar to the investor in the real stock market, each user of this system has two accounts, one is a security account and the other is a bank account. By security, we mean tradable financial asset, such as bond, stock, or future, etc. Since the security account is not allowed to save money, it must be combined with a bank account which provides the money to support the transactions. The user can use the security account to buy and sell securities, and use the bank account to finish the normal banking transactions. When a user tries to buy a security, the system will check whether the combined bank account has enough balance to support this transaction. If the balance is enough, the system will deduct the necessary amount from the bank account and finish the transaction. Otherwise, the system will deny this transaction. In the sell transaction, the system will credit the amount gained from selling the security to the combined bank account.

We constructed both the informal and formal specifications of the system. In the informal specification, we described five first-level functions of the system, including "Deposit", "Withdraw", "Show Balance", "Buy", and "Sell". Each of these five functions is decomposed to several lower level functions. In addition to the functions, four data resources and twelve constraints are specified.

The informal specification was formalized into a formal specification containing nine processes, as shown in Figure 8.1. In order to evaluate the performance of our inspection approach, we inserted 70 bugs into the formal specification. The subjects of our experiment are required to inspect the formal specification against the informal specification to find as many bugs as they can. Table 8.1 summarizes the requirement items and the inspection targets in the informal and formal specifications, respectively.

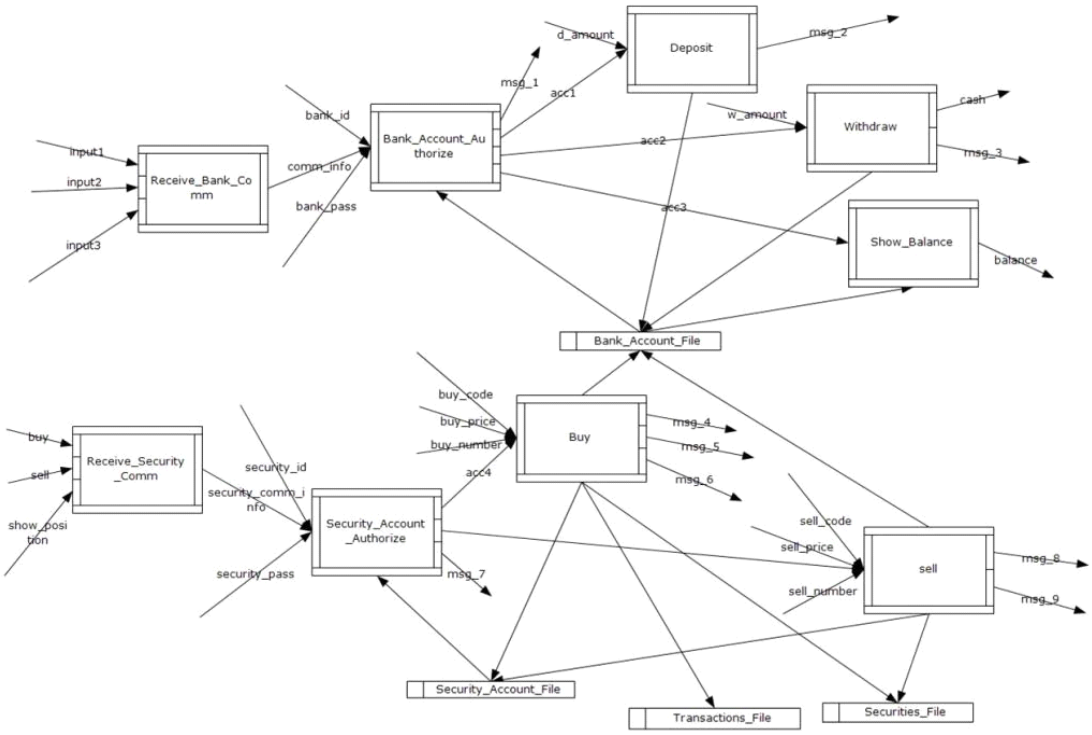


Figure 8.1: CDFD of the traget formal specification

Table 8.1: The requirement items and inspection targets in the experiment

Informal		**Formal**	
Requirement Item	**Number**	**Inspection Target**	**Number**
function	28	system scenario	11
data resource	4	operation scenario	22
constraint	12	(state variable, type)	3
(function, data resource)	22	invariant	14
(function, constraint)	26	(operation scenario, state variable)	74
(data resource, constraint)	8	(operation scenario, invariant)	23
		(state variable, invariant)	8

Table 8.2: The three groups in the experiment

Group	**Person Number**	**Experience in SOFL**	**Inspection Method**
Group A	19	Good experience	Traditional checklist based inspection
Group B	18	Good experience	Our proposed animation based inspection
Group C	18	Normal experience	Our proposed animation based inspection

8.1.2 Subjects

All of the subjects who are invited to our experiment have studied the SOFL three-step development approach, but have different experience in writing SOFL formal specifications. For example, some of the subjects have studied the SOFL for two to three years. These students can understand the SOFL formal specification and the process to develop software system using SOFL. They have written several SOFL formal specifications in related lectures and projects. There is a group of subjects who do not have much experience in SOFL. In the SOFL class in this semester, they have been taught the SOFL development approach and required to construct SOFL informal and formal specifications for two different systems. Moreover, all the subjects who join the SOFL class have been taught the inspection technique and have done some class exercise to inspect the specifications of each other.

In our experiment, the subjects are divided into three groups based on their experience on SOFL. Every group is required to inspect the same formal specification described in the previous section but using different inspection methods. One inspection method used in the experiment is the inspection method proposed in this dissertation, which using animation as reading technique and utilizing traceability to build specific checklist for each inspection target. The other inspection method is the traditional checklist-based inspection method. In this method, the checklist is the only guidance to guide the inspector to check the formal specification, and the abstract questions on the checklist is suitable for the entire specification.

Table 8.2 shows how the subjects are divided and the number of subjects in each group. As shown in the table, the students with good experience in SOFL are divided into "Group A" and "Group B", where by "good experience" we mean the experience in using the SOFL language more than one year. These two groups are required to use the two different inspection methods to inspect the formal specification. We make such arrangement since we want to compare the performance of our inspection method with the other method to evaluate its effectiveness.

As mentioned previously, the inspection mainly conducted by human, different inspection methods can provide different guidance to the inspector but cannot make decision for him. In order to compare the performance of the two different inspection methods, the inspectors who use the different methods should be at the same level. They must have similar experience in using SOFL formal specifications; otherwise, the credibility of the comparison will be affected. The reason we choose the traditional checklist-based inspection method is that it is the most frequently used inspection method in industry [99] [100].

The subjects with normal experience in SOFL (i.e., experienced using SOFL less than half year) are divided into "Group C" and they are required to inspect the formal specification by using our inspection approach. This will allow us to compare the result of Group C and that of Group A to see whether the systematic guidance provided by our inspection method can help offset the subjects' lack of experience in SOFL and to what extent.

8.1.3 Categories of Bugs

Since the major concern of our inspection approach is to check the consistency between the informal and formal specifications, we roughly divide the 70 inserted bugs into two categories: consistency and inside. By consistency, we mean the consistency between the informal and formal specifications. If a bug breaks the consistency between the two specifications, it will be included in the "Consistency" category. Note that whether a bug is a "Consistency" bug cannot be formally defined. In principle, if a bug violates the informal specification directly, it is considered as a "Consistency" bug. If a bug cannot be categorized as "Consistency" bug, it is realized as an "Inside" bug which only affect the correctness and internal consistency of the formal specification. We separate the bugs in this dimension since the traceability between the informal and formal specifications is adopted in our inspection method and we intend to verify whether the traceability related questions can help the inspector detect more bugs that affect the consistency between these two specifications.

As shown in Table 8.3, the bugs in each category are further separated into more detailed classifications. Here we use specific bugs as examples to introduce each bug classification respectively

1. Undefined state variable (USV) (Number of bugs: 1)

   Since four data resources are described in the informal specification, four state variables should be defined in the formal specification to formalizes the data resources. In the target formal specification, we deleted one

Table 8.3: The detailed classifications of bugs

Category	**Classification**	**Number**	**Total**
Consistency	Undefined state variable (USV)	1	30
	Inappropriate invariant definition (IID)	3	
	Undeclared state variable (UDSV)	4	
	Inappropriate state variable declaration (ISVD)	4	
	Inappropriate process definition (IPD)	18	
Inside	Declaring with inappropriate type (DWIT)	11	40
	SOFL-specific bugs (syntax, or logical expressions) (SSB)	10	
	Using inconsistent variable (UIV)	10	
	Using undefined items of types (UUIT)	7	
	Using incorrect state variable name (UISVN)	2	

```
process Deposit(acc1: Bank_Account, d_amount: real)msg_2: string
pre    true
post   acc1.balance = acc1.balance + deposit_amount or msg_2 = “Deposit Success!”
end_process
```

Figure 8.2: The process “Deposit” in the target formal specification with bugs

```
process Withdraw(acc2: Bank_Account, w_amount: real) cash: real | msg_3: string
ext    rd #Bank_Account_FIle
pre    true
post   acc2.balance <= withdraw_amount and acc2.balance = acc2.balance + w_amount
       and cash = w_amount or acc2.balance <= withdraw_amount and msg_2 = “Balance
       Is Not Enough!”
end_process
```

Figure 8.3: The process “Withdraw” in the target formal specification with bugs

```
process Receive_Bank_Comm(input1: string | input2: string | input3: string) comm_info:
Security_Comm_Info
pre   true
post  input1 = “deposite” and comm_info = <deposite> or input2 = “withdraw” and
        comm_info = <withdraw> or input3 = “show_balance” and comm_info =
        <show_balance>
end_process
```

Figure 8.4: The process “Receive_Bank_Comm” in the target formal specification with bugs

of the state variables, therefore the undefined state variable is one bug in the specification.

2. Inappropriate invariant definition (IID) (Number of bugs: 3)

   The invariants are defined to formalize the constraints described in the informal specification, in the target formal specification, we defined three inappropriate invariants. For example, the constraint “*C3.2* ” requires that “*ID number of each bank account should be 7 digital numbers*”, but in the formal specification, we define the corresponding invariant as “*forall[a: Bank Account] | len(a.id) = 4*”. The invariant contains a bug since it does not satisfy the requirement.

3. Undeclared state variable (UDSV) (Number of bugs: 4)

   The state variables used in a process must be declared in front of the pre-condition of the process. Whether a state variable should be used by the process should be decided according to the informal specification. If the process realizes a function that uses a data resource, the process must use the state variable that formalizes the data resource. If the necessary state variable is not declared in the process specification, there is a bug. For example, according to the informal specification, the process “*Deposit*” shown in Figure 8.2 should use the state variable “*Bank_Account_File*”, which presents the database of all bank accounts. But in the target formal specification, we remove the declaration of “*Bank_Account_File*” from the process “*Deposit*”. Therefore, a bug exists in the “*Deposit*” process specification.

4. Inappropriate state variable declaration (ISVD) (Number of bugs: 4)

In the declaration of a state variable, the key words "*wr*" and "*rd*" are used to indicate whether the state variable will be changed by the process or read only by the process, respectively. In the target formal specification, we insert some inappropriate state variable declarations to the process specifications. For example, the process "*Withdraw*" shown in Figure 8.3 defines the function to allow the user to withdraw money from his bank account. The state variable "*Bank_ Account_ File*" is used in the process. Since the process needs to change the state variable after the withdraw being successfully finished, the state variable should be declared with key word "*wr*". But in the target specification, the state variable is declared with key word "*rd*". That means the declaration of the state variable is inappropriate.

5. Inappropriate process definition (IPD) (Number of bugs: 18)

   The inappropriate definitions are the bugs that affect the functionality of the processes. This kind of bug can be caused by using inappropriate logical operators. For example, based on the functionality of process "*Deposit*" shown in Figure 8.2, the two predicates "*acc1.balance = acc1.balance + deposite_ amount*" and "*msg_ 2 = 'Deposit Success!'*" should be conjunctively connected by using key word "*and*". But in the target formal specification, these two predicates are disjunctively connected by using key word "*or*", and therefore the desired functionality of process "*Deposit*" is not appropriately defined. In addition to using inappropriate operators, the IPD bugs can also be caused by missing important functionality. For example, the constraint "*C3.4* " in the informal specification requires that "*Maximum withdraw amount is 300,000*". But in the definition of process "*Withdraw*" shown in Figure 8.3, this constraint is not formalized, and therefore a bug exists in the process specification.

6. Declaring with inappropriate type (DWIT) (Bugs number: 11)

   This kind of bugs declare the variables with inappropriate types in the variable lists of processes. For example, as shown in Figure 8.4, the output variable of process "*Reveive_ Bank_ Comm*" is "*comm_ info*", which is used to transfer the command information to the succeeding process "*Bank_ Account_ Authorize*". The correct type of "*comm_ info*" should be "*Bank_ Comm_ Info*", but in the target formal specification, we declare the variable "*comm_ info*" with inappropriate type "*Security_ Comm_ Info*".

7. SOFL-specific bugs (SSB) (Bugs number: 10)

In the SOFL formal specification, all the variables are transferred as values and there is no reference variable. Moreover, the predicates in pre- and post-conditions only describe the conditions that need to be satisfied and cannot be executed. For example, the predicate "*acc1.balance = acc1.balance + deposit_amount*" in the post-condition of process "*Deposit*" is trying to state that the "*balance*" of "*acc1*" should be updated after the execution of "*Deposit*", and the updated value should equal to "*acc1.balance + deposit_amount*". If the predicate is a statement in the programming languages, like Java or C#, it is defined correctly and the value of the "*balance*" will be changed after the execution of the statement. But in the SOFL formal specification, the predicate is defined incorrectly, the value of "*balance*" cannot be updated by using the predicate, instead, the designer should use SOFL built-in method "*modify*" to update the value of "*balance*".

8. Using inconsistent variable (UIV) (Bugs number: 10)

   By using inconsistent variables, we mean the variables used in the pre- and post-conditions are not consistent with the variables declared in the variable list of the process. For example, the variable "*w_amount*" is declared in the input variable list of process "*Withdraw*" shown in Figure 8.3, however, both the variable "*w_amount*" and "*withdraw_amount*" are used in the post-condition. Althought the variable "*withdraw_amount*" is not declared in the variable list, it represents the same meaning as the variable "*w_amount*". We consider the variables that have features like "*withdraw_amount*" as inconsistent variables.

9. Using undefined items of types (UUIT) (Bugs number: 7)

   The types defined in the SOFL formal specification may have more than one items. For example, the type "*Bank_Comm_Info*" is defined as a "*enumeration*" type and contains three values: "<*deposit*>", "<*withdraw*>", and "<*balance*>". In the process "*Reveive_Bank_Comm*", the variable "*comm_info*" should be declared with type "*Bank_Comm_Info*" and have one of its three values, However, the last predicate in the post-condition of the process is "*comm info = <show_balance>*", and the value "<*show_balance*>" is not one of the values of the type "*Bank_Comm_Info*".

10. Using incorrect state variable name (UISVN) (Bugs number: 2)

    As mentioned above, if a state variable needs to be used in a process, it must be declared in front of the precondition of the process. In the declaration, the name of the state variable must be used correctly,

Table 8.4: The categroies of bugs

Dimension	**Category**	**Number of Bugs**	**Total Number of Bugs**
From domain point of view	Bank	33	70
	Security	37	

otherwise the declaration is considered incorrect. In the target formal specification, we declared two state variables with incorrect names.

In addition to dividing the inserted bugs in the consistency dimension, we also separate the 70 bugs in the domain dimension. Table 8.4 shows the further division of the domain dimension into two categories and the number of bugs in each category.

As introduced in the previous section, the system described in the formal specification includes two parts. One part is used to support the security transaction, and the other part is used to support the banking transaction. As shown in Table 8.4, we separate the bugs that belong to different domains. We divide the bugs in this dimension because we believe that whether the inspector has the domain knowledge of the system described in the formal specification can significantly affect the inspection results. Since the subjects of this experiment are all students, they must have domain knowledge in banking transaction but may not have domain knowledge in security transaction. The number of bugs in each category that can be found in the inspection can reveal the extent of effect caused by domain knowledge.

8.2 Experiment Implementation

In the experiment, each subject is given two hours to do the inspection. Different documents are distributed to the subjects based on the inspection method they use. The details of the documents distributed to each group are described as follows.

8.2.1 The Documents for Traditional Checklist-based Inspection

Since the subjects in Group A are asked to use the traditional checklist-based inspection method to check the formal specification, three kinds of documents are distributed, including the informal specification, the formal

specification, and a checklist. The formal specification includes the CDFD and associated module specification.

The questions on the distributed checklist are abstract questions. For instance, one of the questions on the checklist is "*Whether function is appropriately formalized in the formal specification?*". This question reminds the inspector to check whether the function described in the informal specification is appropriately realized in the formal specification, but it does not contain any information to indicate what part of the formal specification should be checked to answer this question. We built the checklist in this way based on the following three major concerns:

1. The traditional checklist-based inspection method is usually used to check informal specifications, therefore, the questions on the checklist provided to Group A are asked from the informal specification perspective.

2. The questions used in the traditional checklist-based inspection method are abstract questions, therefore, we did not provide any information of the formal specification in the checklist.

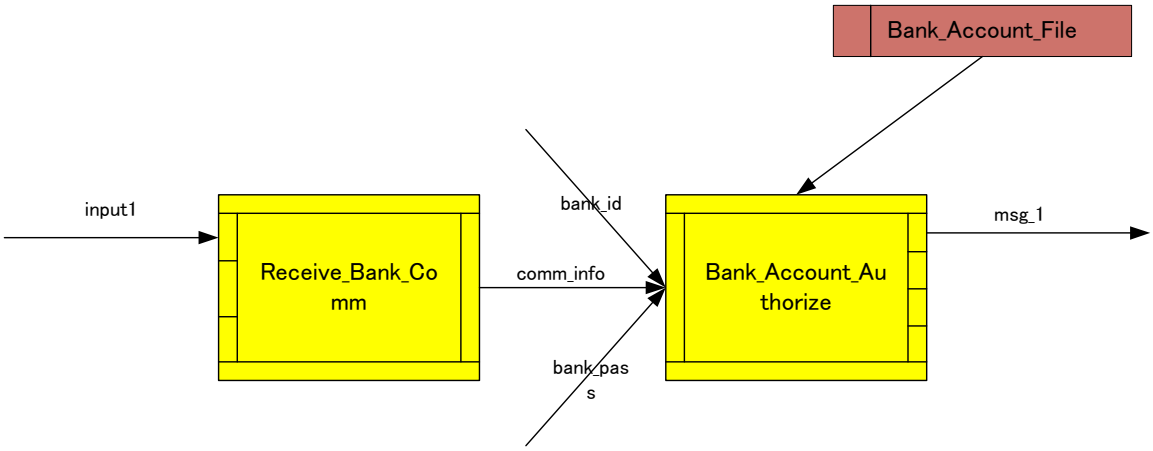
3. In this experiment, the traditional checklist-based inspection method is used to compare with our inspection method. Since the traceability-based checklist is one of the major portions of our inspection method, the checklist used in the traditional checklist-based inspection should not overlap with our traceability-based checklist.

In the inspection, the subjects are required to mark the place where they think there is a bug. The subjects are asked to find as many bugs as they can, and we calculate how many bugs they have found after the experiment.

8.2.2 The Documents for IBSAT

The subjects in Group B and C are asked to use our inspection approach to inspect the formal specification and two documents are distributed. One is the informal specification and the other is a document that combines the formal specification and the checklist. The informal specification distributed to Group B and Group C is the same informal specification we distributed to Group A. The formal specification that we provide to Group B and Group C is prepared based on the procedure of our inspection method and the functions provided by our framework.

In our inspection approach, the inspector reads through the formal specification by following the animation process, therefore, the formal process specifications provided to Group B and Group C are reorganized according



1. Questions for process “Receive_Bank_Comm”

process Receive_Bank_Comm(input1: **string**) comm_info: **Security_Comm_info**
pre true
post input1 = “deposit” and comm_info = <deposit>
end_process

Q1: Whether the name “**Receive_Bank_Comm**” is appropriate?

Q2: Whether the name and type of input variable “**input1**” are appropriate?

Q3: Whether the name and type of output variable “**comm_info**” are appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
input1	string
comm_info	Security_Comm_Info

Q8: Whether the invariants related to process "**Receive_Bank_Comm**" are defined correctly?

Q9: What constraints in the informal specification are formalized by these invariant?

Figure 8.5: The document used in the experiment

2. Questions for process "Bank_Account_Authorize"

process Bank_Account_Authorize(bank_id: **string**, bank_pass: **string**, comm_info: **Security_Comm_info**) msg_1: **string**
pre true
post exists![x: Bank_Account_File] | (x.id = bank_id) and msg_1 = "Error!"
end_process

Q1: Whether the name "**Bank_Account_Authorize**" is appropriate?

Q2: Whether the name and type of input variables "**bank_id**", "**bank_pass**", and "**comm_info**" are appropriate?

Q3: Whether the name and type of output variable "**msg_1**" is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
Bank_Account_File	Bank_Accounts
x.id	nat0
bank_id	string

Q8: Whether the data store "**Bank_Account_File**" is declared appropriately with "**wr**" or "**rd**"?

Q9: What data resources in the formal specification is formalized by the data store "**Bank_Account_File**"?

Q10: Whether the invariants related to process "**Bank_Account_Authorize**" are defined correctly?

```
forall[a: Bank_Account] | len(a.id) = 4;
forall[a: Bank_Account] | len(a.password) = 7;
forall[a, b: Bank_Account] | a <> b => a.id <> b.id;
```

Q11: What constraints in the informal specification are formalized by these invariant?

Figure 8.6: The document used in the experiment (contiune)

to the animation process. We first use the supporting tool to generate the system scenarios from the CDFD shown in Figure 8.1. Then, we prepare the checklist for each derived system scenario. For example, Figure 8.5 and 8.6 show the document used in the experiment for inspecting the first system scenario in the CDFD.

The system scenario that needs to be checked is *{input1}[Receive_Bank_Comm*$_{11}$*, Bank_Account_Authorize*$_{11}$*] {msg_1}*, and two processes are involved in this system scenario. According to our inspection method, the inspector should check the process specifications of "*Receive_Bank_Comm*$_{11}$" and "*Bank_Account_Authorize*$_{11}$" sequentially. Therefore, we provide two independent checklists for the two processes respectively. The checklist shown in Figure 8.5 is used to inspect the process "*Receive_Bank_Comm*$_{11}$", and the checklist shown in Figure 8.6 is used to examine the process "*Bank_Account_Authorize*$_{11}$". In each checklist, we first provide the formal specification related to the process, and then give a list of questions for inspecting the process. The subjects in the experiment are asked to answer each question and mark the place where they think there is a bug.

The questions in the checklist are modified to be suitable to be done on the paper document. The modification is made based on the following principles:

1. The questions should be independent to each other.

   By independent, we mean that a question should not be formed based on the answers of the previous questions. In the supporting tool, the answers of the previous questions will be recorded and reused to construct the following questions based on the built-in question templates. However, in the paper document, the questions are provided in advance, and the answers of the previous questions cannot be reused to construct new questions. For example, one of the questions in the original checklist is "*What functions in the informal specification are formalized by this process?*". If the answer is "*f*", then the answer "*f*" will be reused to form the question "*Whether the name of the process reflects the essential idea of the function 'f '*?". But in the paper document, the answer "*f* " is only a reminder to the subjects that the functions "*f*" should be considered when answering other questions.

2. The questions should be easily understood by the subjects.

   Science the subjects in our experiment are students, they do not have much experience in analyzing formal specifications. Therefore, the questions in the checklist should be formed to be easily understood. For example, we avoid to use the term "*operation functional scenario*" to form the questions since it is the concept used to analyze the formal specification. Instead, we use the term "*process*" to form the questions to help the subject understand the checklist.

3. The scale of the checklist should be limited.

   According to [53], the questions in a checklist should not be more than one single page. We compress the number of the questions and only raise the typical questions from traceability and the four aspects introduced in Chapter 6. Moreover, we eliminate some long and complicated questions since they affect the readability of the checklist. For example, in the supporting tool, the question "*In the formal specification, function 'f ' uses the data resource 'd', whether the process uses the state variable that formalizes the data resource 'd'?*" can be formed based on the answers of the previous questions. But in the paper document, the same question needs to be asked as "*Whether the process uses the state variable that formalizes the data resource used by*

its original requirement function described in the informal specification?" Obviously, the latter question is ambiguity and will affect the readability of the checklist.

The paper document distributed to Group B and C is prepared by complying with the underlying principle of our inspection approach. As mentioned in Chapter 3, the two major portions of our inspection approach are the animation-based reading technique and the traceability-based checklist. Although the animation cannot be performed on the paper document, we reorganize the formal specification so that the subjects in the experiment can read the specification by following the animation procedure. We also provide checklists to each process involved in the animation, and the questions are raised based on the traceability and the four aspects, according to the proposed inspection approach. **Because the purpose of this experiment is to evaluate the performance of our inspection method, using paper document to conduct the experiment does not affect the experiment results and their analysis.**

The major reason we use paper document rather than the supporting tool to conduct the experiment is that using paper document is easy for the subjects marking the bugs. And the marks on the paper document can help us to further analyze our inspection approach. For example, a subject may write something or mark on the question when he tries to answer the question. Such kind of information can be used to help us improve the questions used in the inspection. Note that using paper document does not mean the framework is not important. Actually, the framework is extremely important since it provides the functions to automatically prepare the contents and information included in the paper document, and it saves a lot of effort.

Another reason we do not use the supporting tool is that the subjects in the experiment are not familiar with the animation and inspection functions provided by the tool. The inspection function of the tool provides an interactive manner to raise questions, and it is difficult for the subjects to adapt to the tool in the limited time. Using paper document to perform the experiment can make it more efficient.

8.3 Results Analysis

Based on the marks made by the subjects, we count the bugs found by each subject. In addition to the bugs we inserted to the formal specification, we find that the subjects also discovered some bugs contained in the target formal specification but not inserted by us. The new discovered bugs are all security-related inside bugs, including

2 DWIT bugs, 1 UUIT bug, and 10 UIV bugs. To distinguish with the inserted bugs, the new discovered bugs are called "*inherited bugs*". Table 8.5, 8.6, and 8.7 list up the inspection results of each subject in different groups, respectively, the number of bugs detected by each subject includes both the inserted bugs and inherited bugs.

We include the inherited bugs found by each subject in the final result because the detected inherited bugs took the opportunity cost of the subject to detect inserted bugs. This means that the subject used his time and effort that can be used to detect the inserted bugs for detecting the inherited bugs. Therefore, using both the inserted and inherited bugs to calculate the results is more appropriate.

Since Table 8.5, 8.6, and 8.7 have the same structure, we only explain the data in Table 8.5 and the data in 8.6, and 8.7 can be understood in the same way.

Totally 19 raws of data are included in Table 8.5, each raw represents one subject in Group A. The second column in the table is the total number of bugs found by each subject, including both inserted and inherited bugs. The number "*83*" in the head of this column indicates that totally 83 bugs are included in the target formal specifications (70 inserted bugs and 13 inherited bugs found by the subjects). From the third column to the twelfth column list the number of bugs found by the subjects based on the bug classifications explained in the previous section. Each bug classification is indicated by its abbreviation as introduced in Table 8.3. The number under each abbreviation is the total number of the bugs of each classification. Note that the total numbers of the bugs of the classifications DWIT, UUIT, and UIV are different from the numbers introduced in Table 8.3. This is because the number of inherited bugs that belong to these three classifications are included. For example, we inserted 11 DWIT bugs into the target formal specification, and 2 inherited DWIT bugs are found by the subjects, therefore, the total number of DWIT bugs is 13 (11 + 2), as shown in the eighth column in the table.

In this section, we first analyze the experiment results from macrolevel, namely, comparing the average results between different groups, and then analyze why some bugs are not found by using our inspection approach.

8.3.1 Comparison between Different Groups

Table 8.8 summarizes the inspection results and relevant statistics for each group. The second column in Table 8.8 lists the **average number of bugs** that have been found by the subjects in each group. The third column indicates the **percentage of the bugs** found by each group. The forth and fifth columns in the table are the

Table 8.5: The detailed results made by the subjects in Group A

No	**Total**	**Consistency**					**Inside**				
		USV	**IID**	**UDSV**	**ISVD**	**IPD**	**DWIT**	**SSB**	**UIV**	**UUIT**	**UISVN**
	(83)	**(1)**	**(3)**	**(4)**	**(4)**	**(18)**	**(13)**	**(10)**	**(20)**	**(8)**	**(2)**
1	30	0	2	2	2	5	5	0	8	6	0
2	25	0	3	1	3	2	3	0	11	2	0
3	13	0	2	1	2	0	2	0	5	1	0
4	37	0	3	3	2	6	9	0	11	2	1
5	35	0	0	4	4	5	3	0	18	1	0
6	4	0	0	0	0	2	0	0	2	0	0
7	34	0	3	1	4	7	6	0	10	3	0
8	30	0	0	0	4	2	4	0	15	4	1
9	39	0	3	2	3	6	9	0	14	2	0
10	26	0	3	0	0	3	7	0	13	0	0
11	20	0	3	1	2	3	0	0	9	2	0
12	8	0	0	0	0	2	2	0	4	0	0
13	16	0	3	0	4	4	2	0	3	0	0
14	6	0	1	1	0	1	0	0	3	0	0
15	30	0	0	0	0	5	4	0	15	6	0
16	32	0	2	1	4	3	12	0	8	2	0
17	14	0	0	0	0	3	4	0	6	1	0
18	20	0	0	1	4	5	4	0	5	1	0
19	15	0	0	1	2	5	5	0	1	1	0

Table 8.6: The detailed results made by the subjects in Group B

No	**Total**	**Consistency**					**Inside**				
		USV	**IID**	**UDSV**	**ISVD**	**IPD**	**DWIT**	**SSB**	**UIV**	**UUIT**	**UISVN**
	(83)	**(1)**	**(3)**	**(4)**	**(4)**	**(18)**	**(13)**	**(10)**	**(20)**	**(8)**	**(2)**
1	39	0	3	4	4	8	7	0	12	1	0
2	25	0	0	2	4	5	5	0	7	2	0
3	25	0	2	3	1	4	6	0	7	2	0
4	30	0	3	4	4	7	6	0	6	0	0
5	45	0	1	2	4	6	9	0	16	7	0
6	42	0	3	4	4	7	9	0	10	5	0
7	43	0	3	2	4	7	8	0	12	7	0
8	28	0	2	2	4	4	6	0	6	4	0
9	48	0	3	4	2	11	6	0	15	6	1
10	32	0	3	3	2	6	5	0	10	3	0
11	31	0	2	3	4	5	7	0	9	1	0
12	28	0	2	2	4	2	2	0	12	4	0
13	44	0	3	4	4	6	6	0	16	5	0
14	28	0	3	2	3	3	5	0	10	2	0
15	48	0	3	2	4	8	9	0	17	5	0
16	23	0	3	2	4	4	2	0	7	1	0
17	39	0	3	4	4	6	8	0	9	5	0
18	30	0	3	1	4	4	4	0	10	4	0

Table 8.7: The detailed results made by the subjects in Group C

No	**Total**	**Consistency**					**Inside**				
		USV	**IID**	**UDSV**	**ISVD**	**IPD**	**DWIT**	**SSB**	**UIV**	**UUIT**	**UISVN**
	(83)	**(1)**	**(3)**	**(4)**	**(4)**	**(18)**	**(13)**	**(10)**	**(20)**	**(8)**	**(2)**
1	13	0	2	1	3	2	2	0	3	0	0
2	21	0	2	3	2	4	3	0	6	1	0
3	14	0	2	1	1	1	2	0	7	0	0
4	16	0	2	2	0	1	4	0	6	1	0
5	15	0	2	2	1	2	3	0	5	0	0
6	17	0	2	2	1	3	3	0	6	0	0
7	16	0	2	0	0	3	5	0	6	0	0
8	16	0	2	2	2	1	4	0	5	0	0
9	30	0	2	2	1	5	6	0	12	2	0
10	8	0	2	1	3	1	0	0	1	0	0
11	13	0	0	2	3	0	4	0	4	0	0
12	22	0	3	1	3	3	1	0	9	2	0
13	21	0	1	2	2	1	5	0	9	1	0
14	22	0	1	1	1	5	4	0	8	2	0
15	16	0	0	2	1	3	3	0	7	0	0
16	11	0	2	0	1	3	2	0	3	0	0
17	16	0	0	1	2	0	7	0	4	2	0
18	12	0	0	1	0	2	6	0	3	0	0

Table 8.8: The inspection results of each group and revelant statistics

Group	**Average Number of Bugs (A)**	**Percentage of Bugs (A / 83 × 100%)**	**Variance**	**Standard Deviation**
Group A	22.8	27.5%	112.9	10.6
Group B	34.9	42%	68.3	8.3
Group C	16.6	20%	24.5	4.9

variance and standard deviation each group, which are used to measure the variation or dispersion of the number of bugs found by different subjects in the same group. The higher the value of these two measurements, the larger the difference between two results in the same group.

For instance, the subjects in Group A found 22.8 bugs on average. These bugs are 27.5 percentage of the 83 total bugs (70 inserted bugs + 13 inherited bugs). The variance of the number of bugs found by the subjects in Group A is 112.9, and the standard deviation is 10.6.

Comparing the numbers in the second column of Table 8.8, we can find the subjects in Group B found 53.1% ((34.9 - 22.8) / 22.8 × 100%) more bugs than the subjects in Group A on average. Since the subjects in these two groups are at the same level, we believe that our inspection method is more effective than the traditional checklist-based inspection method. This advantage may result from the fact that our inspection method provides a systematic and practical reading technique and a traceability-based checklist with specific questions. The reading technique and the checklist are combined together to provide the inspector with specific instructions and necessary information that can help them to make judgement. In contrast, the traditional checklist-based inspection method only provides a checklist with abstract questions and does not provide any guidance to instruct the inspector to read through the specification or use the checklist. Therefore, we make the following conclusion:

Conclusion 1 *Our inspection method is more effective than the traditional checklist-based inspection method to help the inspector to detect the defects in the formal specification.*

The numbers in the second column of Table 8.8 also indicates that the subjects in Group B found 110.2% ((34.9 - 16.6) / 16.6 × 100%) more bugs than the subjects in Group C on average. Since the subjects in these two groups use the same inspection method, we believe that this result is mainly caused by the difference of subjects'

experience in SOFL formal specification. As mentioned previously, the subjects in Group B have better experience in SOFL than the subjects in Group C, therefore, we believe that the inspector with better experience will do better in an inspection.

Another evidence also supports this conclusion. This evidence is that the subjects in Group A found 37.3% ((22.8 - 16.6) / 16.6 × 100%) more bugs than the subjects in Group C on average. In the experiment, the subjects in Group A have better experience than the subjects in Group C, but according to Conclusion 1, the subjects in Group C used a more effective inspection method. The result indicates that the advantage in experience helps the subjects in Group A to make better result than the subjects in Group C even the subjects in Group C used a more effective method.

Moreover, we find that using a more effective inspection method can help to offset the difference caused by the lack of experience. This is because Group B found 110.2% more bugs than the subjects in Group C but Group A found only 37.3% more bugs. Merely according to the data collected from the experiment, we believe that using our inspection method can offset 66.1% of the difference caused by the lack of experience, which is calculated using the following formula:

$$\frac{(Bugs\ found\ by\ Group\ B - Bugs\ found\ by\ Group\ C) - (Bugs\ found\ by\ Group\ A - Bugs\ found\ by\ Group\ C)}{(Bugs\ found\ by\ Group\ B - Bugs\ found\ by\ Group\ C)} \times 100\%$$

Although this number is not enough to conclude in what extent our inspection method can help to offset the difference caused by the lack of experience, it does indicate that using our inspection method can help to offset the difference.

Based on the above observations, we can get the following conclusion:

Conclusion 2 *The experience of an inspector can affect the result of an inspection. The inspector with more experience will make better result, but using a more effective inspection method can help to offset the difference caused by the lack of experience.*

The last two columns in Table 8.8 is the variance and standard deviation of the results of each group. Variance and standard deviation measure the variation or dispersion of the number of bugs found by different subjects in a group from the average number of bugs found by the group. If the variance of a group is relatively large, it means the difference between two results in this group is relatively large, and each result is far away from the average

result of this group. Therefore, a relatively small variance indicates that each individual subject in the group make similar result and the result is close to the average result of the group.

As shown in Table 8.8, the variance and standard deviation of Group A are much larger than those of Group B and C. That means comparing the Group A, the subjects in Group B and C found similar number of bugs to other subjects in the same group. Since the subjects in the same group have similar experience in SOFL and they are divided only based on their experience, we think the variation of subjects' abilities of each group should be similar. By ability, we mean the capability to use knowledge and experience to finish a specific task.

Therefore, we believe that the subjects in Group B and C used a more systematic inspection method that leads to the result that the subjects in the same group find similar number of bugs. This is reasonable because our inspection method provides more detailed instructions than the traditional method. For example, we find that some subjects in Group A ignored the bugs inserted to invariants. But in our inspection method, the invariants are considered as inspection targets. The subjects in Group B and C rarely ignored these bugs since specific questions about each invariant are asked in the checklist.

Based on the reasons discussed above, we make the following conclusion:

Conclusion 3 *Regardless of the variation of the abilities of the individual inspectors with similar experience, using our inspection method can ensure that the inspection results of different inspectors are closer than using the traditional checklist-based inspection method.*

The target formal specification used in our experiment contains two parts that are related to different domains. One part describes the functions used to support banking transactions and the other describes the functions for supporting security transactions. Table 8.9 summarizes the inspection results that are related to these two domains. The second column in the table lists the number of bugs related to the banking transactions that have been found, and the third column presents the percentage of the found bugs. The forth and fifth columns record the number of bugs related to security domain that have been found and the percentage of the found bugs, respectively. Note that the total number of the security-related bugs is 50, this is because we inserted 37 security-related bugs to the target formal specification and all 13 inherited bugs are security-related.

As indicated by the data in Table 8.9, the subjects found more banking related bugs than security related bugs,

Table 8.9: The number of bugs found in domain dimension

Group	**Average Number of Bugs of Bank (B)**	**Percentage of Bugs (B / 33 × 100%)**	**Average Number of Bugs of Security (S)**	**Percentage of Bugs (S / 50 × 100%)**
Group A	11	33.3%	11.8	23.7%
Group B	19.72	59.8%	15.2	30.34%
Group C	13.28	40.2%	3.3	6.67%

Table 8.10: The number of bugs found in consistency dimension

Group	**Average Number of Bugs of Consistency(C)**	**Percentage of Bugs (C / 30 × 100%)**	**Average Number of Bugs of Inside (I)**	**Percentage of Bugs (I / 53 × 100%)**
Group A	8.2	27.4%	14.6	27.5%
Group B	14.6	48.5%	20.3	38.3%
Group C	6.7	22.2%	9.9	18.7%

no matter what inspection method they use and what experience they have. We believe that the major cause of this situation is that the subjects have better domain knowledge in banking transactions than in security transactions. Almost all the subjects have a bank account to manage their money, but they rarely have security account to buy and sell security in the stock exchange. Therefore, we make the following conclusion:

Conclusion 4 *An inspector can make better result in an inspection if the target formal specification describes a system in a domain he is familiar with, regardless of what inspection method he uses.*

As mentioned previously, we also divide the bugs within the consistency dimension. In this dimension, bugs are separated into two categories: consistency and inside. The consistency bugs are the bugs that directly affect the consistency between informal and formal specifications, and the inside bugs are the bugs that merely affect the formal specification itself. Table 8.10 summarizes the inspection results that are related to these two categories. Note that the total number of the inside-related bugs is 53, this is because we inserted 40 inside-related bugs to the target formal specification and all 13 inherited bugs are inside-related.

As we can see from Table 8.10, the subjects in Group B found 77.3% ((14.6 - 8.2) / 8.2 × 100%) more consistency

Table 8.11: The summary of Group B under detailed bug classification

	Total (83)	**USV (1)**	**IID (3)**	**UDSV (4)**	**ISVD (4)**	**IPD (18)**	**DWIT (13)**	**SSB (10)**	**UIV (20)**	**UUIT (8)**	**UISVN (2)**
Average	34.9	0	2.5	2.8	3.6	5.7	6.1	0	10.6	3.6	0.06
Percentage	42%	0%	83%	69.5%	88.9%	31.8%	47%	0%	52.8%	44.4%	2.8%

related bugs than the subjects in Group A. Since the subjects in Group B and A have similar experience in SOFL, we believe that using our inspection method helps the subjects in Group B find more consistency bugs. This is reasonable because the checklist used in our inspection method is based on the traceability between informal and formal specifications.

We also find that the subjects in Group C found less consistency related bugs than the subjects in Group A even the subjects in Group C used our inspection method. We believe this is because the subjects in Group C do not have as much experience as the subjects in Group A. According to our Conclusion 2, this situation is not abnormal. However, we find that the subjects in Group A find 22.4% ((8.2 - 6.7) / 6.7 × 100%) more consistency related bugs than the subjects in Group C. If we do not separate the bugs into different categories, subjects in Group A find only 37.3% more bugs in total than the subjects in Group C. This also proves that our inspection method helps the inspector find more consistency related bugs to some extent. Therefore, we make the following conclusion based on our observation:

Conclusion 5 *Our inspection method can help inspector find more consistency-related bugs than the traditional checklist-based inspection method.*

8.3.2 The Unfound Bugs

As shown in Table 8.8, the best result was made by Group B, who found 42 % of total bugs contained in the target formal specification by using our inspection approach. Table 8.11 summarizes the data in Table 8.6. The first raw in Table 8.11 is the average numbers of bugs found in each bug classification, and the second raw is the percentage of the bugs that have been found. In this section, we will analyze why some bugs have not been found by using our inspection approach based on the detailed bug classifications.

1. Undefined state variable (USV) (Number of bugs: 1; Found: 0; Found Percentage: 0%)

   In the experiment, this bug was not found by any subject. We believe that one possible reason is that we asked the subject to follow the animation process and checklist to inspect the formal specification. However, the type definitions and the state variable definitions of the target formal specification are not provided in the checklist directly. Instead, they are provided in an independent document. If answering a question needs the definitions of state variables, the subject needs to refer to the independent document himself. Since we did not give the direct instruction to require the subject to refer to the independent document, the subject may not check the state variable definitions. In this case, the bug can not be found.

2. Inappropriate invariant definition (IID) (Number of bugs: 3; Found: 2.5; Found Percentage: 83%)

   The inspection result indicates that most of the subjects in Group B found the 3 IID bugs. We think one possible reason is that the invariant is considered as an inspection target in our inspection approach, and the invariants related to each process are provided in the checklist. Such kind of checklist can help the subject find IDD bugs.

3. Undeclared state variable (UDSV) (Number of bugs: 4; Found: 2.8; Found Percentage: 69.5%)

   The result indicates that most of the subjects in Group B can find at least 2 UDSV bugs. The other 2 UDSV bugs that have not been found by some subjects maybe because our checklist does not include the question "*Whether the process uses the state variable that formalizes the data resource used by its original requirement function described in the informal specification?*".

   In the ideal procedure, the subject should refer to the informal specification to answer the question "*What functions in the informal specification are formalized by the process?*" and then refers to the related data resources in the informal specification based on the answer of the question. The related data resources can remind the subject whether all necessary state variables are declared. However, not all the subjects can follow this ideal procedure because the procedure was taught to them as part of training but not provided in the checklist. If a subject cannot follow the ideal procedure, he has to make the judgement based on his knowledge and experience. But there are 2 UDSV bugs related to the security domain, which is an unfamiliar domain to the subjects, the subjects may not be able to find the bugs. The following case provides an example

```
process Buy(acc4: Security_Account, buy_code: string, buy_price: real, buy_number: nat)
msg_6:
ext rd #Bank_Account_File
    wr #Security_Account_File
    wr #Securities_File
pre   true
post  exist![x: Security_Account_File and y: Bank_Account_File and z: Securities_File] |
       (x.id = acc4.id and y.security_account_id = acc4.id and z.code = buy_code and
       buy_price < z.bid and msg_6 = "The Offered Price is Too Cheap!")
end_process
```

Figure 8.7: Part of the process "Buy" in the target formal spcecification with bugs

of the UDSV bug that has not been found.

Case 1: *The process "Buy" in the formal specification describes the function to allow the user to buy a security. According to the informal specification, the information of a "buy" transaction should be recorded after the transaction is successfully executed. Therefore, the process "Buy" should use the state variable "Transactions_File", which is used to record all transactions. However, the state variable "Transaction_File" was not declared in the process specification of "Buy". Some subjects failed to find this bug since they are not familiar with the security transactions.*

4. Inappropriate state variable declaration (ISVD) (Number of bugs: 4; Found: 3.6; Found Percentage: 88.9%)

   Since we provide an question in the checklist to ask the subject to check whether the state variables are declared with appropriate key word "*wr*" or "*rd*", most of the subjects found all 4 ISVD bugs. One possible reason that some subjects missed the ISVD bugs is that they are not careful enough. Since the question is only a reminder to the subjects to help them to find bugs, the subjects need to be careful enough to answer each question and make decisions.

5. Inappropriate process definition (IPD) (Number of bugs: 18; Found: 5.7; Found Percentage: 31.8%)

   Finding IPD bugs is the most difficult. The subject needs to fully understand the functionality of each process to make judgement. The inappropriate definition can be caused by many reasons, such as using

inappropriate operators, or using inappropriate variables. The following case presents an IPD bug caused by using inappropriate operator and not found by some subjects.

Case 2: *In the process "Withdraw" shown in Figure 8.3 provides the function to allow the user to withdraw money from his bank account. After the withdraw transaction is successfully finished, the balance in the bank account is deducted by the amount that the user just withdrew. In the process "Withdraw", the predicate "acc2.balance = acc2.balance + w_amount" is defined to describe the function of updating the bank account. In this predicate, the variable "acc2 " refers to the bank account and the variable "w_amount" presents the desired amount the user wants to withdraw. Since a success withdraw transaction should deduct money from the bank account, the operator "+" is used inappropriately. In the experiment, some subjects failed to find this bug. One possible reason is that these subjects did not read the process specification carefully. All the subjects must be familiar with the withdraw transaction, if they can read the specification carefully enough, they would find this obvious bug.*

Another kind of IPD bug that is rarely found by the subjects is missing functionality. The following case illustrates an IPD bug that is rarely found.

Case 3: *The constraint "C3.4 " in the informal specification requires that "Maximum withdraw amount is 300,000". But in the definition of process "Withdraw" shown in Figure 8.3, this constraint is not formalized, and therefore the definition of process "Withdraw" is inappropriate. Actually, this bug can be detected by answering the question asked for checking the relations between functions and constraints described in the informal specification. But we eliminated such question due to the readability concern. In order to detect this bug, the subjects must notice the constraint "C3.4 " in the informal specification, otherwise, this bug cannot be detected.*

Compared to the above two cases that are related to banking transaction, finding the IPD bugs that related to security transaction is more difficult. The following case gives an example of an unfounded IPD bug that is related to security transactions.

Case 4: *The target formal specification describes a security transaction system that allows its user to buy or*

```
process Bank_Account_Authorize(comm_info: Security_Comm_Info, bank_id: string,
bank_pass: string)msg_1: string | acc1: Bank_Account | acc2: Account | acc3: Account
pre true
post (exists![x: Bank_Account_File] | (x.id = bank_id) and (bank_comm_info = <deposite>
      and acc1 = x or bank_comm_info = <withdraw> and acc1 = x and bank_comm_info =
      <show_balance> and acc3 = x)) or (exists![x: Bank_Account_File] | (x.id = bank_id)
      and msg_1 = "Error!")
end_process
```

Figure 8.8: The process "Bank_Account_Authorize" in the target formal specification with bugs

sell securities to a broker. The broker maintains two prices of a security, one is the price ("bid" price) to buy the security from the user and the other is the price ("ask" price) to sell the security to the user, respectively. The price that the broker uses to sell the security is called "ask" price, and the "ask" price is the cheapest price that an user can buy the security from the broker. If a user want to buy a security, he needs first offer a price to the broker. The system will check whether the offered price is lower than the "ask" price. If the offered price is lower, the buy transaction cannot be executed. The portion of process "Buy" shown in Figure 8.7 describes the situation that the offered price is lower than the "ask" price. However, as shown in Figure 8.7, the variable "buy_price", which presents the offered price, is compared to the variable "z.bid ". It is a bug since the offered price is compared to an inappropriate price. There may be two reasons that the subjects did not find the bug. One is that the subjects need to read the informal specification, including the functions, data resources, and constraints to understand the functionality described in process "Buy". The other reason is that the subjects do not have related domain knowledge to help them make judgement.

The other unfounded IPD bugs have similar features as the above three cases, and the reasons that they are not found can be explained in the same way.

6. Declaring with inappropriate type (DWIT) (Bugs number: 13; Found: 6.1; Found Percentage: 47%)

   In our checklist, questions are raised to ask the subjects to check whether the types of the input and output variables of a process are appropriate. To check whether a DWIT bug exists, the subjects need to understand

the functionality of the process and refer to the type definitions. Since the type definitions are provided as an independent document in the experiment, the subjects need to refer to the type definitions by themselves. Based on our analysis of the inspection results, the most possible reason that some DWIT are not found is that the subjects did not refer to the type definitions. For example, the process "*Bank_ Account_ Authorize*" shown in Figure 8.8 contains 2 DWIT bugs. The output variable "*acc2*" and "*acc3*" of the process are declared with inappropriate types because the type "*Account*" is not defined in the target formal specification. In the experiment, some subjects failed to find these 2 DWIT bugs. We believe the possible reason is that they did not refer to the type definitions.

7. SOFL-specific bugs (SSB) (Bugs number: 10; Found: 0; Found Percentage: 0%)

   The reason that no SOFL-specific bug was found in the experiment is that the subjects are not familiar with the grammar of SOFL. The subjects used the grammar of programming languages to understand the SOFL formal specification in the experiment. For example, the "*equality symbol*" in the programming language means assignment, but in SOFL formal specification, the "*equality symbol*" is a logical operator. The sentence in the program that contains "*equality symbol*" is an assignment sentence, and it can be executed to change the value of the variable on the left side of the "*equality symbol*". But in the SOFL formal specification, a logical expression containing "*equality symbol*" can not be executed and it only describes a condition that the value of the variables on the two sides of the "*equality symbol*" operator should be equal. If a designer wants to change the value of a compound type variable, he should use the built-in functions such as "*modify*".

8. Using inconsistent variable (UIV) (Bugs number: 20; Found: 10.6; Found Percentage: 52.8%)

   Using inconsistent variables is the most frequently made mistake in writing formal specifications. However, we did not provide a question to ask the subjects to check whether all variables used in the process are consistent with the declared variables in the variable list. The subjects are expected to detect the UIV bugs when they read the formal specification. In the experiment, we found that whether an UIV bug can be found is dependent on the individual subject, and some UIV bugs will be ignored without intention if they do no affect the understanding of the process. For example, the variable "*w_ amount*" is declared in the input variable list of process "*Withdraw*" shown in Figure 8.3, however, both the variable "*w_ amount*" and

"*withdraw_ amount*" are used in the post-condition. Since the inconsistent variable "*withdraw_ amount*" is used twice in the post-condition, there are two UIV bugs existed in the process "*Withdraw*". If a subject can find one of the two UIV bugs, that means he has the ability to find the other one since the two bugs are caused by the same reason. However, we found that some subjects detected only one of the two UIV bugs. Based on the observation, we cannot conclude that the subjects cannot find the UIV bugs. One possible explanation is that the subjects ignored some UIV bugs without intention.

9. Using undefined items of types (UUIT) (Bugs number: 8; Found: 3.6; Found Percentage: 44.4%)

   The subjects need to refer to the type definitions to detect a potential UUIT bug. Usually, detecting UUIT bugs are relatively easy. As long as the subjects refer to the original type definitions, the UUIT bugs can be detected. One possible reason that more than half of the UUIT bugs were not found is that we did not provide an explicit instruction to ask the subjects refer to the type definitions and therefore they did not do so for detecting UUIT bugs.

10. Using incorrect state variable name (UISVN) (Bugs number: 2; Found: 0.06; Found Percentage: 2.8%)

    Although our checklist provides questions to check whether the state variables are declared with appropriate key words, there is no question to ask the subjects to check the names of state variables. The subjects may ignore the UISVN bugs if there is no instruction to ask them to do so.

8.3.3 Possible Improvements

Based on the analysis in the previous section, the following possible improvements can be adopted to make our inspection approach more effective.

1. Integrating defect-based questions to the checklist

   Defect-based questions can help the inspector detect the bugs based on their categories. For example, as explained previously, the IPD bugs can caused by many kinds of defects and are very difficult to be detected. Using defect-based questions can help the inspector to find a specific kind of defect that causes the IPD bugs. For example, the question "*Whether all operators used in the pre- and post-conditions are used appropriately?*"

can be asked. This question provides very detailed instruction to ask the inspector to check each operator in the process.

Using defect-based questions to perform inspection is usually called defect-based reading [75]. The questions are formed based on well defined defect categories. Since the classifications of defects in SOFL formal specification have not been well studied, to systematically integrate the defect-based question into our checklist is impossible in the current stage, but it will be our future work.

2. Building a type checker

   A type checker is very important for constructing high quality formal specifications. It can be used to automatically detect the UIV bugs, UUIT bugs, UISVN bugs, and part of the DWIT bugs. In that case, the inspector can pay more attention to the detection of consistency-related bugs contained in the formal specifications.

3. Using supporting tool to perform the inspection

   In our inspection approach, using questions to check the relations between different requirement items is an important feature. But we eliminated such questions in the checklist due to the readability concern. The elimination of such questions makes the detection of some kinds of bugs to be difficult. For example, the bugs in Case 1 presented in the previous section may be found by checking the relation between the process and the state variable, and the bug in Case 3 may be detected by examining the relation between the function and the constraint. By using the supporting tool, questions for checking relations between two different requirement items can be formed automatically based on the constructed trace links. Therefore, we believe that using supporting tool to perform the inspection can make better results.

4. Asking the inspector checking the formal specification from the SOFL perspective

   Since the formal specification is not program, the inspector should not understand the formal specification in the same way he understands the program. In order to detect the formal specification language related defects, the inspector must read the specification from the formal specification perspective.

Although we can improve our inspection approach by integrating defect-based questions and other techniques, we should notice that the questions in the inspection are only reminders to the inspector, whether the inspector can be patient and careful enough to answer each question on the checklist can significantly affect the inspection result.

8.4 Findings in the Experiment

In addition to the conclusions made in the previous section, we also find some points based on the experience of the experiment. The major findings as listed as follows:

1. Our inspection method does provide a good reading technique for inspection.

The interview of some subjects after the experiment indicates that we provide a good reading technique in our inspection method. The animation can guide the inspector to read through the entire formal specification and it can make inspection targets in each step very clear.

2. Some questions are asked several times.

In our method, the inspection is carried out by following the animation process. Since the same processes may be animated for several times in the animation, the related inspection targets of these processes may be inspected for several times. For example, there are two system scenarios in the experiment contain same processes:

$$\{\text{input2}\}[\text{Receive\_Bank\_Comm}_{21}, \text{Bank\_Account\_Authorize}_{13}, \text{Withdraw}_{11}]\{\text{cash}\}$$

$$\{\text{input2}\}[\text{Receive\_Bank\_Comm}_{21}, \text{Bank\_Account\_Authorize}_{13}, \text{Withdraw}_{12}]\{\text{msg\_3}\}$$

To inspect the above system scenario, the inspection targets related to "Receive_ Bank_ Comm_{21}" and "Bank_Account_Authorize_{13}" need to be inspected twice. Answering the same questions will take more time in inspection. However, we find an advantage of such duplication. That is if a subject cannot find the bugs when he first inspect an inspection targets, he may find the bugs in the second time when he inspects the same inspection target.

3. Inviting the inspector to join the specification construction process is likely to benefit the inspection.

Whether the inspector understands the contents and structure of the specification affects the inspection result. Therefore, inviting the inspector to join the specification construction process or asking the designer who writes the specification to conduct the inspection is an cost-effective way to improve the performance of an inspection.

We list four major findings we have found from this experiment, and all of them should be considered in our future research and in future inspection practice.

8.5 Threats

The major threats that have potential to affect the validity of the experiment are listed as follows:

1. Lack of sufficient training

Lack of sufficient training may affect the effectiveness of the inspection. Although we have taught the students about the inspection technique in the class, the training may not be sufficient to allow them to have enough experience and skills as the industrial professionals in using inspection methods to detect defects in the formal specification. This threat can however be mitigated by extending the training time to let them gain more experience.

2. Lack of experience

The experience of students in using SOFL to write formal specifications for software systems is not comparable to experienced professionals in industry. Lack of experience will restrict the students in understanding the structure and contents of the formal specification. This may affect the result of the inspection. An advantage of using students is that we can assess whether our inspection method can be used by inexperienced practitioners in industry. This threat can be compensated by conducting a larger experiment involving industrial practitioners in the future.

3. Lack of understanding of our inspection method

Since the students may not have enough experience in constructing SOFL specifications, understanding our inspection method may be difficult for them. The major concept in our method that may confuse the students is the traceability between informal and formal specifications. To understand the traceability, the students must clearly understand how the informal specification is evolved to the formal specification. For example, we build a trace link between a function to a group of operation scenarios, but the students usually formalize a function into

one process. They need more experience to understand why the function is formalized as a group of operation scenarios. Without a clear understanding of the traceability, the students may be confused by the questions on the checklist and therefore may not find bugs based on the questions.

In spite of the threats mentioned above, the experiment reported in this chapter has helped us evaluate our inspection method to a certain extent. We are interested in using more realistic and large scale formal specifications to rigorously assess our inspection method in the future.

8.6 Summary

In this Chapter, we presented an experiment for evaluating the performance of our inspection method. The results of the experiment indicated that our inspection method can help the inspector find more bugs in the formal specification than the traditional checklist-based inspection method, especially the bugs related to the consistency between the informal and formal specifications. We also pointed out some findings we have found in this experiment, and these findings should be paid more attention in inspection practice. In the next Chapter, we will discuss the work related to our research and compare the differences.

Chapter 9

Related Work

The inspection approach presented in this dissertation is proposed for formal specification validation. It contains an animation-based reading technique and a checklist of questions raised based on the traceability between informal and formal specifications. In this chapter, we discuss the related work in the relevant research areas and indicate the difference and contribution of our inspection approach.

9.1 Animation

Animation is a technique to dynamically present the behaviors of formal specifications. It gives the user and expert a chance to observe the operational behaviors of the specification and enhance their confidence before the specification is implemented. After extensively studying of the literature, we find that several techniques have been developed for animating or dynamically presenting formal specifications, however, no work has explored the same approach as we propose in this dissertation to utilize specification animation as a reading technique to assist specification inspection.

As far as specification animation is concerned, most of the existing work we have discovered take the approach that first transforms the formal specification into executable code written in a programming language and then animate the specification by executing the code. PiZA [101] is an animator for Z formal specification. It translates Z specifications into Prolog to generate outputs. Similarly, an animation approach for Object-Z specification is described in [102], which translates the Z specification into C++ code for execution. Morrey *et al.* developed a tool called "wiZe" to support the construction of model-based specification language, in particular Z, and the animation of an executable subset of Z specification language [103]. The subtool for animation of specifications is called ZAL. The wiZe is responsible for making a syntactically correct specification and transforming it into

an executable representation in an extended LISP, and then passes the executable representation to ZAL. ZAL animates the specification by executing the specification with test cases.

PVS (Prototype Verification System) [45] is a specification and verification system including an expressive specification language and interactive theorem prover. It includes a ground evaluator [104] that can translate an executable subset of PVS to Common Lisp and provides a read-eval-print loop for testing the translated Lisp program. In [105], the PVS is enhanced from two points for evaluation and animation purposes. One enhancement is the extension of the ability of executing PVS specification. A technique called "semantic attachment" is introduced. It allows the user to provide their own executable program segments to different PVS functions, therefore some PVS specification that cannot be translated by the ground evaluator can be executed. And another enhancement is to animate the PVS specification by displaying the results of ground evaluator in the Graphical User Interfaces (GUIs) created by Tcl/Tk.

In [68], Liu and Wang introduced an animation tool called SOFL Animator for SOFL specification animation. It can be used to perform syntactic and semantic analysis of a specification. When performing an animation, the tool will automatically translate the executable subset of SOFL specification into Java program segments, and then use some test case to execute the program. In order to provide a graphic presentation of the operational behaviors of specification, SOFL Animator uses Message Sequence Chart (MSC) to present the execution of the translated Java program.

MSC is also adopted in other animation approach as a framework to provide a graphical user interface to represent animation. Stepien and Logrippo built a toolset to translate LOTOS traces to MSC and provide a graphic animator [106]. The translation is based on the mappings between the elements of LOTOS and MSC. Combes and his colleagues described an open animation tool for telecommunication systems in [107]. The tool is named as ANGOR, and it offers an environment based on a flexible architecture. It allows animating different animation sources, such as formal and executable language like SDL and scenario languages like MSC.

VDMTools is an industry-strength toolset supporting the analysis of system models expressed in VDM [70] and has been successfully used in several industrial projects [108] [109] [110]. The interpreter inside the VDMTools can execute a large executable subset of VDM specification. User can test the VDM specification by providing test cases and observe the system behavior by setting breakpoints or stepping. Moreover, the interpreter in VDMTools

can automatically create an external logfile recording all the events happened during an execution. In this logfile, each event is tagged by the time at which it occurs, this information can be used to graphically display all the events on a time-axis. Overture [111] provides functions similar to VDMTools and is built on an open and extensible platform based on the Eclipse framework. By using the combinatorial testing function [112], a large collection of test cases can be executed to detect the run-time errors caused by forgotten pre-condition or violation of invariant and post-condition.

ProB [69] [113] is a validation toolset for B-Method. It can automatically check the consistency of B specification via model checking. The model checker in ProB explores the state space of a specification and determine whether a state violate an invariant. It can graphically display the path from an initial state to a counter-examples when a violation of invariant is discovered. However, in order to perform the exhaustive model checking, the given sets must be finite, and the integer variables must be restricted to a small range. The animation of B specification is carried out by separating the model checking process. Start from the initial state, in each step of the animation, the user can choose an action to proceed to the next state.

Another tool that adopts model checking for presenting the dynamic behavior of systems is UPPAAL [72] [114], which allows user to model the system behavior in terms of states and transitions between states. Unlike Z, B, or VDM, there is no article specification in UPPAAL. User can set invariants to each state, and set guards and action to each transition. The invariants describe the condition that should be satisfied by the state; the guards restrict the possible state changes by disabling transitions; and the actions change the value of variables. To animate the model, the *simulator* of UPPAAL can explore the state space of the model in a step-by-step fashion.

Time Miller and Paul Strooper introduced a framework for animating model-based specifications by using testgraphs [67]. The framework provides a testgraph editor for users to edit testgraphs, and then derive sequences for animation by traversing the testgraph. Gargantini and Riccobene proposed an automatically driven approach to animating formal specifications in Parnas' SCR tabular notation [115]. One important feature of this work is the adoption of a model checker to help find counter-examples that contain a state not satisfying the property to be established by animation. The advantage of this work is that the formal specification can be checked by means of model checking in the animation.

Some of the existing studies, like PVS ground evaluator, wiZe, and SOFL Animator, require an automatic

translation of the formal specification into an executable programming language. This will inevitably limit the capability of animation because specifications using pre- and post-conditions may not be automatically transformed into code in general. Therefore, what these animation tools can do is only to deal with a subset of the formal notations mentioned above. In contrast to this situation, our animation approach described in this dissertation does not require any translation of the specification into code; it can directly perform animation by evaluating the pre- and post-conditions of the processes involved in the target system scenarios for the selected test cases and expected results. This is much easier to implement technically and capable of dealing with all pre-post style specifications. VDMTools and Overture also evaluate the pre- and post-conditions but in the run-time of executing test cases. However, only a large subset of VDM specification can be executed and explicit specification must be constructed for execution. The explicit specification is an algorithm solution of an operation in VDM. The specification contains only pre- and post-conditions cannot be executed. The animation tools based on model checking, like ProB and UPPAAL, have inherent challenge of state explosion, the states of the system must be finite. Although our animation approach has similarities with some exiting studies, no exiting work adopts animation as a reading technique to guide the user read through the specification for inspection purpose.

9.2 Inspection

Inspection is a static analysis technique used to verify and validate any artifacts produced in difference software development phase including requirement analysis, design, and code. The concept of inspection was first introduced by Fagan at IBM [17]. Fagan's inspection is performed by a team following a well defined process. The inspection team is organized by a moderator who manages the team and coordinates the inspection process. Other team members include author, reader, and tester. The author is the person who produced the target artifact in the inspection; the reader paraphrases the artifact during the team meeting; the tester reviews the artifact from testing point of view. Fagan required the inspection team follow a six-step process: planning, overview, preparation, inspection meeting, rework, and follow-up. In the preparation step, each team member should inspect the target material independently. During the inspection meeting, the team members discuss the record all the defects. Although the original Fagan inspection method provided a well organized framework for carrying out inspection, it did not indicate a specific technique for the individual inspector to review and detect faults from the target

artifact. The inspection approach proposed in this dissertation is mainly designed to indicate inspector to read through the artifact and find defects.

The inspection meeting is emphasized in Fagan's inspection method, however, some researchers indicated that most of the defects are detected by the inspector individually in the preparation step rather than the team meeting [116] [117] [118]. Several techniques have been developed to help reviewers to accomplish their job in the preparation step. These techniques are called *reading techniques* and they play significant affects to the final result of inspection. Since the reading techniques are used to help the inspector to detect defects, we use "reading technique" and "inspection method" interchangeably in the following of this section. Aurum *et al.* summarized and listed the following reading techniques in [75].

- **Ad hoc reading**

  In Ad hoc reading, there is no instruction or guidance is provided. The target software artifact is simply given to each inspector who has to use (s)he own knowledge and experience to detect faults.

- **Checklist-based reading**

  In checklist-based reading, a list of questions is provided to the inspector. The questions on the checklist specify the problems and properties of the target artifact that need to be checked. It is expected that the checklist will guide the inspector throughout the preparation step. The inspection procedure proposed by Fagan in [17] included the use of checklist.

- **Defect-based reading**

  In defect-based reading, defects of a target artifact are classified and a set of questions is prepared for each defect class. In the meanwhile, scenarios are provided. Each scenario is a procedure aiming at detecting a specific class of defects. Therefore, the defect-based reading is also referred as scenario-based reading.

- **Perspective-based reading**

  The perspective-based reading is an enhanced version of defect-based reading. It focuses on different point of view or needs of stakeholders. Procedures are developed based on the viewpoint of stakeholders. In the preparation step, inspectors read the target artifact from a particular viewpoint and detect relevant defects.

Checklist-based reading technique is usually discussed in requirements engineering books as a representative, and some checklists are also provided [119] [14]. Loosely speaking, the questions in a checklist are only reminders for inspector and often limited to the types of defect that have been detected before. Furthermore, as point in [120], the checklist-based reading technique usually does not provide any instruction about how to use the checklist. There is no guidance for the inspector to indicate when to use what kind of information to answer the questions on the checklist.

Several comparisons have been made between different reading techniques. For example, the perspective-based reading technique was compared with checklist-based reading technique in [121]. The results of individual inspectors indicated that the effectiveness of defect detection using both inspection techniques is similar. L. He *et al* did a similar comparison in [122]. Their results showed that perspective-based reading was significantly more effective than checklist-based reading. However, the comparison in [123] concluded that there is no clear positive effect of perspective-based reading. It was more effective than ad-hoc approaches, but was less effective when compared to checklists.

In the industry, Ad hoc reading and checklist-based reading are used more frequently than other reading techniques. A survey conducted by Sulehri [100] ask the requirement engineers in six software company in Sweden to choose the best requirement validation techniques. Concerning the requirement inspection, the six companies used either the Ad hoc reading or the checklist-based reading. The survey conducted by Saqi [99] about six companies from two countries also indicated that the Ad hoc reading and checklist-based reading are frequently used compared to other reading techniques.

We believe every inspector should answer some questions in the inspection to detect defects. The difference among different inspection methods is that whether the these questions are specified explicitly as a checklist. In our inspection approach, we construct an explicit checklist, similar to the checklist-based reading, to remind inspector what should be checked in the inspection. However, the traditional checklist-based reading only provides a checklist to the inspector without any instruction on how to perform the inspection [124]. In our inspection approach, we provide not only the checklist but also the instructions to help the inspector to read the artifact and carry out the inspection.

The defect-based reading can be realized as a refinement of checklist-based reading since it provides scenarios

to help inspector to implement the items on the checklist [65]. Namely, scenarios are designed as procedures that describe the steps to be accomplished to answer particular questions on the checklist. Even the defect-based reading provides some guidance to guide the inspector read the artifact, it does not contain a systematic instruction to guide the inspector read through the entire artifact. Some contents of target artifact may be missed if the scenarios are not well designed. In our inspection method, the animation-based reading technique are used to guide the inspector read through the entire artifact, every content of artifact would be inspected by following the animation process.

Some researchers proposed other reading techniques. For example, *stepwise-abstraction reading* requires an inspector to read a fragment of code and then abstract the functions implemented by the fragment [125]. *Usage-based reading* in [126] asks the inspector to focus on the most important aspects of software artifact from a point of view of user. An inspection method called "combined-reading technique" is introduced in [66] to inspect requirements specification. The method categorizes the defects of specification and proposes a set of corresponding questions. It requires all the specification should be checked, but it does not indicate a specific technique to easily read through the specification.

As far as formal specification inspection is concerned, there are few researchers who have explored inspection techniques for formal specifications validation. Guido *et al.* [127] introduce an automatic approach to validate pre/postcondition-based specifications, which automatically constructs abstractions in the form of behavior models from the specification. The reviewer validates the behavior model rather than the formal specification itself. Liu *et al.* proposed a rigorous inspection method called RIM in [96]. The focus of this inspection method is the internal consistency of formal specification. RIM defines a group of consistency properties and requires all these properties be satisfied. Compared to the inspection approach put forward in this dissertation, RIM focuses on the verification of internal consistency of formal specifications rather than their validation as our approach aims to deal with. Further, our inspection technique is characterized by adopting specification animation as a reading technique, which is a novel contribution to the field.

9.3 Traceability

The importance of requirement traceability for ensuring system quality is broadly recognized [94] [95] It aims at defining relationships between stakeholder requirements and artifacts produced during the software development

life-cycle [128]. Although we build some traceability rules to clarify the relations between informal and formal specifications, we do not focus on the methodology of constructing traceability. In our inspection approach, the traceability of specifications is used to guide inspector verify and validate the formal specification.

Several researches focus on methodology of building requirement traceability. Pinherio *et al.* [129] proposed a quite formal approach focusing on context requirement tracing, which states that traceability problem caused both by technical and social factors. They built a network to trace dependencies and transitive rules among different factors. Easterbrook *et al.* [130] introduced an AND/OR table as an intermediate to connect textual requirements and SCR model. However, they did not provide a concrete process to generate trace information. George in [131] proposed an approach to automatically detect traceability relations between requirement artifacts and object models using *heuristic traceability rules*. These rules define how to match requirements in the textual artifacts with the items in object models, which have syntactical relation with the requirements. The syntactic relations are defined as patterns of words which have specific grammatical roles in a piece of text.

In [132], Peraldi-Frati et al. divide the requirement traceability into two groups. One is called horizontal traceability, which identifies the relationships across workgroups or components. The other one is vertical traceability, which identifies the connections between the items in different levels of artifacts. Different components in the requirement can be linked by relationships *decompose*, *derive*, or *copy* etc. and the connections between requirement and its solution model (i.e. design or implementation) can be described by the relationships *satisfy*. Moreover, the *verify* relation allows user to link validation and verification elements (such as test cases) to requirements.

Alexander et al. in [128] introduce a tool-supported technique to automatically construct the traceability between requirement and code. The authors require user to provide some test scenarios, which are described as an artifact or a group of artifacts. By testing the scenarios on the program, related implementation classes, methods, and statements of code can be observed. The trace dependencies can be built between the artifacts and related code segment.

A traceability model is proposed in [133]. In this model, the requirements are first imported into a database, and a component of the model called *Traceability Viewer Component* (TVC) makes requirements available to developer as a background service. Developer writes a module or function based on the requirements and insert the related requirements to the module through the TVC. This model also provides a mechanism to validate and

verify whether the requirements are actually being met. A quality assurance specialist can use a component called *Quality Assurance Interface* (QAI) to exam if the code being checked does meet the corresponding requirement.

Jayeeta *et al.* introduced a method to construct traceability between UML diagrams in [134]. Since UML lacks the rigor of formal modeling language, ensuring traceability within UML models becomes difficult. The authors first propose a formal grammar for three UML diagrams: use case diagram, activity diagram, and class diagram. Then, some traceability rules are formally defined for constructing trace links within the three kinds of UML diagrams. However, the authors do not indicate a detailed procedure to build the traceability.

9.4 Summary

In this Chapter, we introduced the existing research related to our work and compared the their difference. The introduction included the three important techniques used in our method: animation, inspection, and traceability. In next Chapter, we will conclude this dissertation and point out the future work.

Chapter 10

Conclusion and Future Work

10.1 Conclusion

Since the quality of requirements specifications can significantly affects the quality of final software products, the techniques used to verify and validate the requirements specifications have become one of the most important topics in software engineering. In this dissertation, we present a novel inspection approach based on specification animation and traceability for formal specification verification and validation.

To guide the inspector reading through a formal specification, an animation method called *system functional scenario-based animation method* is proposed. A system scenario in the animation presents an independent behavior defined in the formal specification. It consists of a sequence of operation scenarios. When animating a specific system scenario, the inspector can inspect each operation scenario involved in turn.

The operation scenarios and related formal specification items that need to be checked are called inspection targets in our approach. We prepare a checklist for each inspection target to remind the inspector what should be checked. Each checklist contains a group of specific questions which are constructed based on trace links. The trace links are the relations between the requirements in the informal specification and the corresponding items in the formal specification. In order to build the trace links, we first formally defined all the requirement items and the inspection targets. Then, we formally defined the traceability rules for building trace links.

For verification and validation purpose, we designed the questions for each inspection target to address four aspects: necessity, appropriateness, correctness, and completeness. We proposed several properties for each aspect to formally define the conditions that should be satisfied by the inspection targets. In the checklist, traceability-related questions are asked first for building the trace links, then, the specific questions raised for these four aspects can be constructed based on the trace links.

To facilitate the inspector, a tool that can support the entire inspection process has been built. The tool not only supports our inspection method, but provides a flexible framework to support the entire SOFL three-step development approach and related techniques, such as pattern-based formal specification construction and automatic predicate evaluation. The tool has been used widely by students in the classes of teaching the SOFL specification language and the related development techniques.

Moreover, we have conducted an experiment to evaluate the performance of our inspection method. The results of the experiment indicate that our approach is a systematic approach and can help the inspector find more bugs than the traditional checklist based inspection method. We also found some findings in inspection practice and weakness of our inspection, which will be the focus of our future research.

10.2 Future work

The future work can be divided into two parts: one is on the inspection method and the other is on the supporting tool.

10.2.1 Research on Inspection Method

According to the experiment results, many questions are asked repeatedly in the inspection process and it makes the entire inspection process time consuming. In order to solve this problem, we intend to optimize the inspection process, namely to dynamically organize the checklist for each inspection target. The questions that have been answered should not appear on the checklist for the rest of the inspection.

The experiment results also indicate that the inspector can find more bugs in the specification that describes a system in their familiar domain. We believe this is because their domain knowledge can help them to understand the functionality of the specification better. Therefore, if we can provide more explanations to the system they are not familiar with, the effectiveness of our inspection method can be improved. For example, we can adopt the visualization technique [135] [136] to graphically present the physical environment and functionality of the system. However, the effectiveness of adopting this technique and other potential techniques needs to be carefully investigated.

The traceability-based checklist in our inspection approach is designed for inspecting the consistency between

informal and formal specifications. However, if a user wants to inspect the formal specification from a different point of view, a new checklist should be designed. Although the questions asked in the inspection may be different, the user can still take the advantage of our reading technique and well defined inspection targets. To this end, we should build a question base to manage the questions proposed by different users. In order to reuse the proposed questions, the question base should allow the user to easily search the questions he interests and automatically construct checklist based on the selected questions.

The traceability-related checklist in our inspection approach helps inspectors examine the formal specification against the informal specification. If the traceability between the formal specification and the implemented program can be constructed, the principle underlying our inspection approach can be used to check the program against the formal specification. In order to extend the current approach for inspecting program, we need to find the general relations between formal specification items and different components in the program. We also need to find the factors that can affect the consistency between the formal specification and the program and design checklist for checking these factors.

The experiment conducted in this dissertation compared our inspection method with the traditional checklist-based inspection method. In the future, we should compare our inspection method to more defect detection methods. Moreover, we plan to invite industry people to evaluate our inspection method in industry settings. Since industry people are usually have different point of view than students, they can provide us with more practical comments. The cost-effectiveness of our inspection method also needs to be checked.

10.2.2 Research on Supporting Framework

At present, our supporting framework has been used in teaching SOFL, and many SOFL projects have been successfully built with its support. During this process, many bugs have been found and fixed, and the usability problems are also solved. Although the current version of framework can provide stable functions to support our research, enhancing its fault tolerance is always the most important task for us. Our final goal is to make this framework mature sufficiently to be used by industry people. To achieve this goal, a lot of testing must be done.

Considering the functionality of the framework, the most urgent function we want is a powerful parser. Although we have integrated a parser built by former master student in our laboratory, it is not powerful enough to satisfy

our requirement. It can only provide simple syntax checking, but we need a parser that can extract necessary information from the formal specification. Fortunately, a student in our laboratory has already started to develop such a powerful parser. In the future, we plan to integrate the parser smoothly into the current framework.

The experiment conducted in this dissertation only evaluates the performance of our inspection method, the usability of the function of the framework used to support the inspection process have not been evaluated. Therefore, another important future research is to invite some students and industry people to evaluate our tool support formal specification inspection method. However, before any experiment or survey can be conducted, the participants must be well trained. Any comments from them will help us to improve the usability of our framework.

References

[1] IEEE Guide for Software Requirements Specifications. *IEEE Std 830-1984*, pages 1–26, Feb 1984.

[2] R. N. Charette. Why software fails [software failure]. *Spectrum, IEEE*, 42(9):42–49, 2005.

[3] J. C. Knight. Safety Critical Systems: Challenges and Directions. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 547–550, May 2002.

[4] C. F. Fan, S. Yih, W. H. Tseng, and W. C. Chen. Empirical Analysis of Software-induced Failure Events in the Nuclear Industry. *Safety science*, 57:118–128, 2013.

[5] R. S. Pressman. *Software Engineering: A PractitionerâĂŹs Approach*. 2005.

[6] W. W. Royce. Managing the Development of Large Software Systems. In *Proceedings of IEEE WESCON*, volume 26. Los Angeles, 1970.

[7] W. M. Wilson. Writing Effective Natural Language Requirements Specifications. *Naval Research Laboratory*, 1999.

[8] A. Aurum and C. Wohlin. *Engineering and Managing Software Requirements*, volume 1. Springer, 2005.

[9] J. Verner, J. Sampson, and N. Cerpa. What Factors Lead to Software Project Failure? In *Research Challenges in Information Science, 2008. RCIS 2008. Second International Conference on*, pages 71–80, June 2008.

[10] R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002.

[11] B. W. Boehm. Verifying and Validating Software Requirements and Design Specifications. In *IEEE software*. Citeseer, 1984.

[12] D.R. Wallace and R.U. Fujii. Software Verification and Validation: An Overview. *Software, IEEE*, 6(3):10–17, May 1989.

[13] J. Radatz, A. Geraci, and F. Katki. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610121990:121990, 1990.

[14] K. E. Wiegers. *Software Requirements*. Microsoft press, 2003.

[15] W. C. Hetzel and B. Hetzel. *The Complete Guide to Software Testing*. QED Information Sciences Wellesley, MA, 1988.

[16] I. Bashir and A. L. Goel. *Testing Object-Oriented Software: Life Cicle Solutionss*. Springer, 2000.

[17] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[18] L. Judith, J. D. Cyrus, and P. D. Harry. Formal Specification and Structured Design in Software Development. *Hewlett-Packard Journal*, December 1991.

[19] M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In *FM 2006: Formal Methods*, pages 147–162. LNCS, Springer, 2006.

[20] P. G. Larsen, J. Fitzgerald, S. Wolff, N. Battle, K. Lausdahl, A. Ribeiro, and K. Pierce. Tutorial for Overture/VDM++. Technical report, 2010.

[21] J. R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[22] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. de Sousa. *Rigorous Software Development: An Introduction to Program Verification*. Springer, 2011.

[23] C. B. Jones. *Systematic Software Development using VDM*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1986.

[24] J. M. Spivey and J. R. Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[25] S. Schneider. *The B-method: An Introduction*. Palgrave Oxford, 2001.

[26] P. Behm, P. Benoit, A. Faivre, and J. A. Meynadier. METEOR: A Successful Application of B in a Large Project. In *FMâĂŹ99âĂŤFormal Methods*, pages 369–387. Springer, 1999.

[27] P. R. Smith and P. G. Larsen. Applications of VDM in Banknote Processing. In *VDM in Practice: Proc. First VDM Workshop*, page 45, 1999.

[28] P. G. Larsen, P. Mukherjee, and K. Sunesen. Using VDMTools to Model and Validate the Cash Dispenser Example. *Formal aspects of computing*, 12(4):216–217, 2000.

[29] I. Houston and S. King. CICS Project Report Experiences and Results From the Use of Z in IBM. In *VDM'91 Formal Software Development Methods*, pages 588–596. Springer, 1991.

[30] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Direction. *ACM Computing Surveys*, 28(4):626–643, 1996.

[31] J. C. Knight, C. L. DeJong, M. S. Gibble, and L. G. Nakano. Why Are Formal Methods Not Used More Widely? In C. M. Holloway and K. J. Hayhurst, editors, *Fourth NASA Langley Formal Methods Workshop*, 3356, pages 1–12, Hampton, Viginia, 1997.

[32] D. L. Parnas. Really Rethinking Formal Methods. *Computer*, 43(1):28–34, January 2010.

[33] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–39, 2009.

[34] S. Liu and R. Adams. Limitations of Formal Methods and An Approach to Improvement. In *Proceedings of 1995 Asia-Pacific Software Engineering Conference (APSEC'95)*, pages 498–507, Brisbane, Australia, Dec 1995. IEEE Computer Society Press.

[35] 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.

[36] S. Liu. *Formal Engineering for Industrial Software Development Using the SOFL Method*. Springer-Verlag, ISBN 3-540-20602-7, 2004.

[37] S. Liu, Y. Chen, F. Nagoya, and J. A. McDermid. Formal Specification-Based Inspection for Verification of Programs. *Software Engineering, IEEE Transactions on*, 38(5):1100–1122, Sept 2012.

[38] W. Miao and S. Liu. A Formal Engineering Framework for Service-Based Software Modeling. *Services Computing, IEEE Transactions on*, 6(4):536–550, Oct 2013.

[39] S. Liu and W. Shen. A Formal Approach to Testing Programs in Practice. In *Systems and Informatics*

(ICSAI), 2012 International Conference on, pages 2509–2515. IEEE, 2012.

[40] C. Tian, S. Liu, and S. Nakajima. Utilizing Model Checking for Automatic Test Case Generation from Conjunctions of Predicates. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 304–309, March 2011.

[41] X. Wang, S. Liu, and H. Miao. A Pattern System to Support Refining Informal Ideas into Formal Expressions. In JinSong Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *Lecture Notes in Computer Science*, pages 662–677. Springer Berlin Heidelberg, 2010.

[42] S. Liu, T. Tamai, and S. Nakajima. Integration of Formal Specification, Review, and Testing for Software Component Quality Assurance. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 415–421, New York, NY, USA, 2009. ACM.

[43] S. Liu. Utilizing Formalization to Test Programs without Available Source Code. In *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, pages 216–221, Aug 2008.

[44] J. Dawes and J. Dawes. *The VDM-SL Reference Guide*, volume 12. Pitman London, 1991.

[45] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Automated DeductionâĂŤCADE-11*, pages 748–752. Springer, 1992.

[46] S. Owre, J. M. Rushby, N. Shankar, and F. Von Henke. Formal Verification of Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[47] S. Liu, M. Asuka, K. Komaya, and Y. Nakamura. An Approach to Specifying and Verifying Safety-Critical Systems with the 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.

[48] H. Hussmann, B. Demuth, and F. Finger. Modular Architecture for a Toolset Supporting OCL. *Science of Computer Programming*, 44(1):51–69, 2002.

[49] A. Hall. Realising the Benefits of Formal Methods. In *Proceedings of 7th International Conference on Formal Engineering Methods*, pages 1–4. LNCS 3785, Springer-Verlag, November 1-4 2005.

[50] N. G. Leveson. Safeware Engineering: Engineering for a Safe World. Technical description, http://www.safeware-eng.com/index.php/white-papers/verification.

[51] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[52] ORA Canada. Z/EVES. Z/eves homepage, http://www.ora.on.ca/z-eves/.

[53] T. Gilb and D. Graham. *Software Inspection*. Addison Wesley, 1993.

[54] NASA. Software Formal Inspections Standard. Technical report, NASA-STD-2202-93, 1993.

[55] D. L. Parnas and M. Lawford. The Role of Inspection in Software Quality Assurance. *IEEE Transactions on Software Engineering*, 29(8):674–676, August 2003.

[56] R. H. Thayer. *Software Reviews, Inspections and Audits: A Standards-based Guide*. Software Engineering Standards Series. IEEE CS Press, June 2010.

[57] D. A. Wheeler, B. Brykczynski, and R. N. Meeson. *Software Inspection: An Industry Best Practice for Defect Detection and Removal*. IEEE Computer Society Press, 1996.

[58] K. E. Wiegers. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley, 2001.

[59] S. Misra, L. Fernández, and R. Colomo-Palacios. A simplified Model for Software Inspection. *Journal of Software: Evolution and Process*, 26(12):1297–1315, 2014.

[60] J.C. Knight and E.A. Myers. Phased Inspections and Their Implementation. *ACM SIGSOFT Software Engineering Notes*, 16(3):29–35, 1991.

[61] S. Kollanus and J. Koskinen. Survey of Software Inspection Research: 1991-2005. Computer science and information systems reports working papers wp-40, University of Jyvaskyla, Finland, 2007.

[62] O. Laitenberger. A Survey of Software Inspection Technologies. In *Handbook of Software Engineering and Knowledge Engineering*, pages 517–556. World Scientific Publishing, 2002.

[63] D. Winkler. *Improvement of Defect Detection with Software Inspection Variants: A Large-Scale Empirical Study on Reading Techniques and Experience*. VDM Verlag, May 2008.

[64] A. A. Porter, H. Siy, C.A. Toman, and L.G. Votta. An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development. *IEEE Transactions on Software Engineering*, 23(6),

June 1997.

[65] A. A. Porter, L. G. Votta, and V. R. Basili. Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment. *IEEE Trans. Softw. Eng.*, 21(6):563–575, June 1995.

[66] A. A. Alshazly, A. M. Elfatatry, and M. S. Abougabal. Detecting Defects in Software Requirements Specification. *Alexandria Engineering Journal*, 53(3):513 – 527, 2014.

[67] T. Miller and P. Strooper. A Framework and Tool Support for the Systematic Testing of Model-Based Specifications. *ACM Transactions on Software Engineering and Methodology*, 12(4):409–439, 2003.

[68] S. Liu and H. Wang. An Automated Approach to Specification Animation for Validation. *Journal of Systems and Software*, (80):1271–1285, 2007.

[69] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FME 2003: FORMAL METHODS, LNCS 2805*, pages 855–874. Springer-Verlag, 2003.

[70] J. Fitzgerald, P. G. Larsen, and S. Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3, 2008.

[71] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

[72] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[73] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publisher, 1993.

[74] Z. Duan, C. Tian, and L. Zhang. A Decision Procedure for Propositional Projection Temporal Logic with Infinite Models. *Acta Informatica*, 45(1):43–78, Feb. 2008.

[75] A. Aurum, H. Petersson, and C. Wohlin. State-of-the-art: Software Inspections after 25 Years. *Software Testing, Verification and Reliability*, 12(3):133–154, 2002.

[76] M. Li and S. Liu. Automated Functional Scenarios-Based Formal Specification Animation. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 107–115, Dec 2012.

[77] M. Li and S. Liu. Traceability-Based Formal Specification Inspection. In *Software Security and Reliability (SERE), 2014 Eighth International Conference on*, pages 167–176, June 2014.

[78] M. Li and S. Liu. Reviewing Formal Specification for Validation Using Animation and Trace Links. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 1, pages 286–293, Dec 2014.

[79] M. Li and S. Liu. Design and Implementation of a Tool for Specifying Specification in SOFL. In Shaoying Liu, editor, *Structured Object-Oriented Formal Language and Method*, volume 7787 of *Lecture Notes in Computer Science*, pages 44–55. Springer Berlin Heidelberg, 2013.

[80] M. Li and S. Liu. SOFL Specification Animation with Tool Support. In Shaoying Liu and Zhenhua Duan, editors, *Structured Object-Oriented Formal Language and Method*, Lecture Notes in Computer Science, pages 118–131. Springer International Publishing, 2014.

[81] 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.

[82] C. L. Ling, W. Shen, and D. Kountanis. Applying SOFL to a Generic Insulin Pump Software Design. In *Structured Object-Oriented Formal Language and Method, 2nd International Workshop SOFL 2012*, pages 116–132. LNCS 7787, Springer, November 2012.

[83] Y. Wang and H. Chen. Extension on Transactional Remote Services in SOFL. In *Structured Object-Oriented Formal Language and Method, 2nd International Workshop SOFL 2012*, pages 133–147. LNCS 7787, Springer, November 2012.

[84] J. Wang, S. Liu, Y. Qi, and D. Hou. Developing an Insulin Pump System using the SOFL Method. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pages 334–341. IEEE, 2007.

[85] W. Miao and S. Liu. Service-oriented Modeling using the SOFL Formal Engineering Method. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pages 187–192. IEEE, 2009.

[86] S. Liu and Y. Sun. Structured Methodology+Object-oriented Methodology+Formal Methods: Methodology of SOFL. In *Engineering of Complex Computer Systems, 1995. Held jointly with 5th CSESAW, 3rd IEEE*

RTAW and 20th IFAC/IFIP WRTP, Proceedings., First IEEE International Conference on, pages 137–144, Nov 1995.

[87] M. Fowler and K. Scott. *UML Distilled: a Brief Guide to the Standard Object Modeling Language (2nd Edition)*. Addison-Wesley, 2002.

[88] URL: http://www.wpcentral.com/ie9-windows-phone-7-adobe-flash-demos-and-development-videos.

[89] URL: http://codereview.stackexchange.com/questions/55767/finding-all-paths-from-a-given-graph.

[90] F. E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[91] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.

[92] S. Liu and S. Nakajima. A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications. In *Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on*, pages 147–155, June 2010.

[93] M. Li and S. Liu. Tool Support for Rigorous Formal Specification Inspection. In *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*, pages 729–734, Dec 2014.

[94] B. Ramesh. Factors Influencing Requirements Traceability Practice. *Communications of the ACM*, 41(12):37–44, 1998.

[95] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou. Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 135–144. IEEE, 2005.

[96] S. Liu, J. A. McDermid, and Y. Chen. A Rigorous Method for Inspection of Model-Based Formal Specifications. *IEEE Transactions on Reliability*, 59(4):667–684, December 2010.

[97] X. Wang and S. Liu. Computer-Aided Formalization of Requirements Based on Patterns. *IEICE TRANSACTIONS on Information and Systems*, 97(2):198–212, 2014.

[98] S. Zhu and S. Liu. A Supporting Tool for Syntactic Analysis of SOFL Formal Specifications and Automatic

Generation of Functional Scenarios. In Shaoying Liu and Zhenhua Duan, editors, *Structured Object-Oriented Formal Language and Method*, Lecture Notes in Computer Science, pages 104–117. Springer International Publishing, 2014.

[99] S. B. Saqi and S. Ahmed. Requirements Validation Techniques Practiced in Industry: Studies of Six Companies. *M. Sc. Thesis, Department of System and Software Engineering, School of Engineering, Blekinge Institute of Technology, Ronneby, Sweden*, 2008.

[100] L. H. Sulehri and G. Bai. Comparative Selection of Requirements Validation Techniques Based on Industrial Survey. *M. Sc. Thesis, Department of Interaction and System Design, School of Engineering, Blekinge Institute of Technology, Ronneby, Sweden*, 2009.

[101] M. Hewitt, C. O'Halloran, and C. Sennett. Experiences with PiZA: an Animator for Z. In *Proceedings of the 1997 Z User Meeting (ZUM'97)*, volume 1212 of *LNCS*, pages 37–51. Springer-Verlag, 1997.

[102] M. Najafi and H. Haghighi. An Animation Approach to Develop C++ Code from Object-Z Specifications. In *Computer Science and Software Engineering (CSSE), 2011 CSI International Symposium on*, pages 9–16. IEEE, 2011.

[103] I. Morrey, J. Siddiqi, R. Hibberd, and G. Buckberry. A Toolset to Support the Construction and Animation of Formal Specifications. *Journal of Systems and Software*, 41(3):147–160, June 1998.

[104] N. S. Shankar. Efficiently Executing PVS. Technical report, Project report, ComputerScience Laboratory, SRI International, Menlo Park, 1999.

[105] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, Testing, and Animating PVS Specifications. *Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep*, 2001.

[106] B. Stepien and L. Logrippo. Graphic Visualization and Animation of LOTOS Execution Traces. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 40(5):665–681, 2002.

[107] P. Combes, F. Dubois, and B. Renard. An Open Animation Tool: Application to Telecommunication Systems. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 40(5):599–620, 2002.

[108] P. G. Larsen and J. Fitzgerald. *Triumphs and Challenges for the Industrial Application of Model-Oriented Formal Methods*. Technical Report Series. University of Newcastle upon Tyne, 2007.

[109] T. Kurita, M. Chiba, and Y. Nakatsugawa. Application of a Formal Specification Language in the Development of the âĂIJMobile FeliCaâĂİ IC Chip Firmware for Embedding in Mobile Phone. In *FM 2008: Formal Methods*, pages 425–429. Springer, 2008.

[110] T. Kurita and Y. Nakatsugawa. The Application of VDM to the Industrial Development of Firmware for a Smart Card IC Chip. *Int. J. Software and Informatics*, 3(2-3):343–355, 2009.

[111] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. *The Overture Initiative Integrating Tools for VDM*. 2010. 2010; 19.

[112] P. G. Larsen, K. Lausdahl, and N. Battle. Combinatorial Testing for VDM. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 278–285, Sept 2010.

[113] M. Leuschel and M. Butler. ProB: An Automated Analysis Toolset for the B Method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, February 2008.

[114] F. Vaandrager. A First Introduction to UPPAAL. *Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook*, 18, 2011.

[115] A. Gargantini and E. Riccobene. Automatic Model Driven Animation of SCR Specifications. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering, 6th International Conference (FASE 2003), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2003)*, volume 2621 of *LNCS*, Warsaw, Poland, April 7-11 2003. Springer-Verlag.

[116] O. Laitenberger, K. El Emam, and T. G. Harbich. An Internally Replicated Quasi-Experimental Comparison of Checklist and Perspective-Based Reading of Code Documents. *IEEE Trans. Softw. Eng.*, 27(5):387–421, May 2001.

[117] P. M. Johnson and D. Tjahjono. Assessing Software Review Meetings: A Controlled Experimental Study using CSRS. In *Software Engineering, 1997., Proceedings of the 1997 International Conference on*, pages 118–127, May 1997.

[118] P. McCarthy, A. Porter, H. Siy, and L. G. Votta. An Experiment to Assess Cost-benefits of Inspection Meetings and Their Alternatives: A Pilot Study. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, pages 100–111, Mar 1996.

[119] I. Sommerville and G. Kotonya. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[120] O. Laitenberger and J-M DeBaud. An Encompassing Life Cycle Centric Survey of Software Inspection. *Journal of Systems and Software*, 50(1):5 – 31, 2000.

[121] G Sabaliauskaite, F. Matsukawa, S. Kusumoto, and K. Inoue. An Experimental Comparison of Checklist-Based Reading and Perspective-Based Reading for UML Design Document Inspection. In *Proceedings of the 2002 International Symposium on Empirical Software Engineering*, ISESE '02, pages 148–, Washington, DC, USA, 2002. IEEE Computer Society.

[122] L. He and J. Carver. PBR vs. Checklist: A Replication in the N-fold Inspection Context. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 95–104, New York, NY, USA, 2006. ACM.

[123] M. Ciolkowski. What Do We Know About Perspective-based Reading? An Approach for Quantitative Aggregation in Software Engineering. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 133–144, Washington, DC, USA, 2009. IEEE Computer Society.

[124] M. Staron, L. Kuzniarz, and C. Thurn. An Empirical Assessment of Using Stereotypes to Improve Reading Techniques in Software Inspections. In *Proceedings of the Third Workshop on Software Quality*, 3-WoSQ, pages 1–7, New York, NY, USA, 2005. ACM.

[125] A. Dunsmore, M. Roper, and M. Wood. The Development and Evaluation of Three Diverse Techniques for Object-Oriented Code Inspection. *IEEE Trans. Softw. Eng.*, 29(8):677–686, August 2003.

[126] T. Thelin, P. Runeson, and C. Wohlin. An Experimental Comparison of Usage-based and Checklist-based Reading. *Software Engineering, IEEE Transactions on*, 29(8):687–704, Aug 2003.

[127] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated Abstractions for Contract Validation. *IEEE Transactions on Software Engineering*, 38(1):141–162, 2012.

[128] A Egyed and P. Grunbacher. Automating Requirements Traceability: Beyond the Record & Replay Paradigm. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 163–171. IEEE, 2002.

[129] F. Pinheiro and J. A Goguen. An Object-oriented Tool for Tracing Requirements. In *Requirements Engineering, 1996., Proceedings of the Second International Conference on*, page 219. IEEE, 1996.

[130] S. Easterbrook and J. Callahan. Formal Methods for Verification and Validation of Partial Specifications: A Case Study. *Journal of Systems and Software*, 40(3):199–210, 1998.

[131] G. Spanoudakis. Plausible and Adaptive Requirement Traceability Structures. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 135–142. ACM, 2002.

[132] M. A. Peraldi-Frati and A. Albinet. Requirement Traceability in Safety Critical Systems. In *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, pages 11–14. ACM, 2010.

[133] A. M. Salem. Improving Software Quality through Requirements Traceability Models. In *Computer Systems and Applications, 2006. IEEE International Conference on.*, pages 1159–1162. IEEE, 2006.

[134] J. Chanda, A. Kanjilal, S. Sengupta, and S. Bhattacharya. Traceability of Requirements and Consistency Verification of UML Use Case, Activity and Class Diagram: A Formal Approach. In *Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on*, pages 1–4. IEEE, 2009.

[135] L. Ladenberger, J. Bendisposto, and M. Leuschel. Visualising Event-B Models with B-Motion Studio. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Proceedings of FMICS 2009*, volume 5825 of *Lecture Notes in Computer Science*, pages 202–204. Springer, 2009.

[136] D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, and M. Leuschel. Validation of the ABZ Landing Gear System using ProB. In *ABZ 2014: The Landing Gear Case Study*, 2014.

Chapter 11

Appendix: Documents Used in the Experiment

Informal Specification

1 Functions

1.1 Deposite
- 1.1.1 Receive command
- 1.1.2 Receive cash
- 1.1.3 Update bank account balance

1.2 Withdraw
- 1.2.1 Receive command
- 1.2.2 Check bank account password
- 1.2.3 Receive withdraw amount
- 1.2.4 Pay cash

1.3 Check balance of bank account
- 1.3.1 Receive command
- 1.3.2 Check bank account password
- 1.3.3 Show balance

1.4 Buy securities
- 1.4.1 Receive command
- 1.4.2 Check securities and bank accounts password
- 1.4.3 Receive security code and number
- 1.4.4 Receive target price
- 1.4.5 Check security price
- 1.4.6 Check bank account balance
- 1.4.7 Make and record transaction, update the balance

1.5 Sell securities
- 1.5.1 Receive command
- 1.5.2 Check securities and bank accounts password
- 1.5.3 Receive security code and number
- 1.5.4 Receive target price
- 1.5.5 Check security price
- 1.5.6 Make and record transaction, update the balance

2 Data Resources

2.1 Bank account database. It is the database of all bank account. Each account should include account number, password, holder's name and balance, and it should connect to a security account for buying and selling securities.(F1.1.2, F1.1.3, F1.2.2, F1.2.3. F1.2.4, F1.3.2, F1.3.3, F1.4.2, F1.4.6, F1.4.7, F1.5.2, F1.5.6)

2.2 Securities account database. It is the database of all security account, each account should contain ID, password, and holder's name. The money for buying and selling securities should be deposit in correspoding bank account(F1.4.2, F1.4.7, F1.5.2, F1.5.6, F1.6.2)

2.3 Securities database. It is the database of all securities. Each security has code, bid price and ask price. Bid price is the most expensive price that an investor can sell the security, and the ask price is the cheapest price that an inverstor can buy the security. Each security also need indicate how many shares are in the stock.(F1.4.5, F1.4.7, F1.5.5, F1.5.6)

2.4 Transactions database. It is the database to record all the transaction, no matter buy or sell.(F1.4.7, F1.5.6)

3 Constraints

3.1 Only six commands can be received: "deposite", "withdraw", "showbalance", "buy", "sell", and "showposition".(F1.1.1, F1.2.1, F1.3.1, F1.4.1, F1.5.1)

3.2 ID number of each bank account should be 7 digital numbers.(F1.2.2, F1.3.2, F1.4.2, F1.5.2, D2.1)

3.3 Password of each bank account should be 4 digital numbers.(F1.2.2, F1.3.2, F1.4.2, F1.5.2, D2.1)

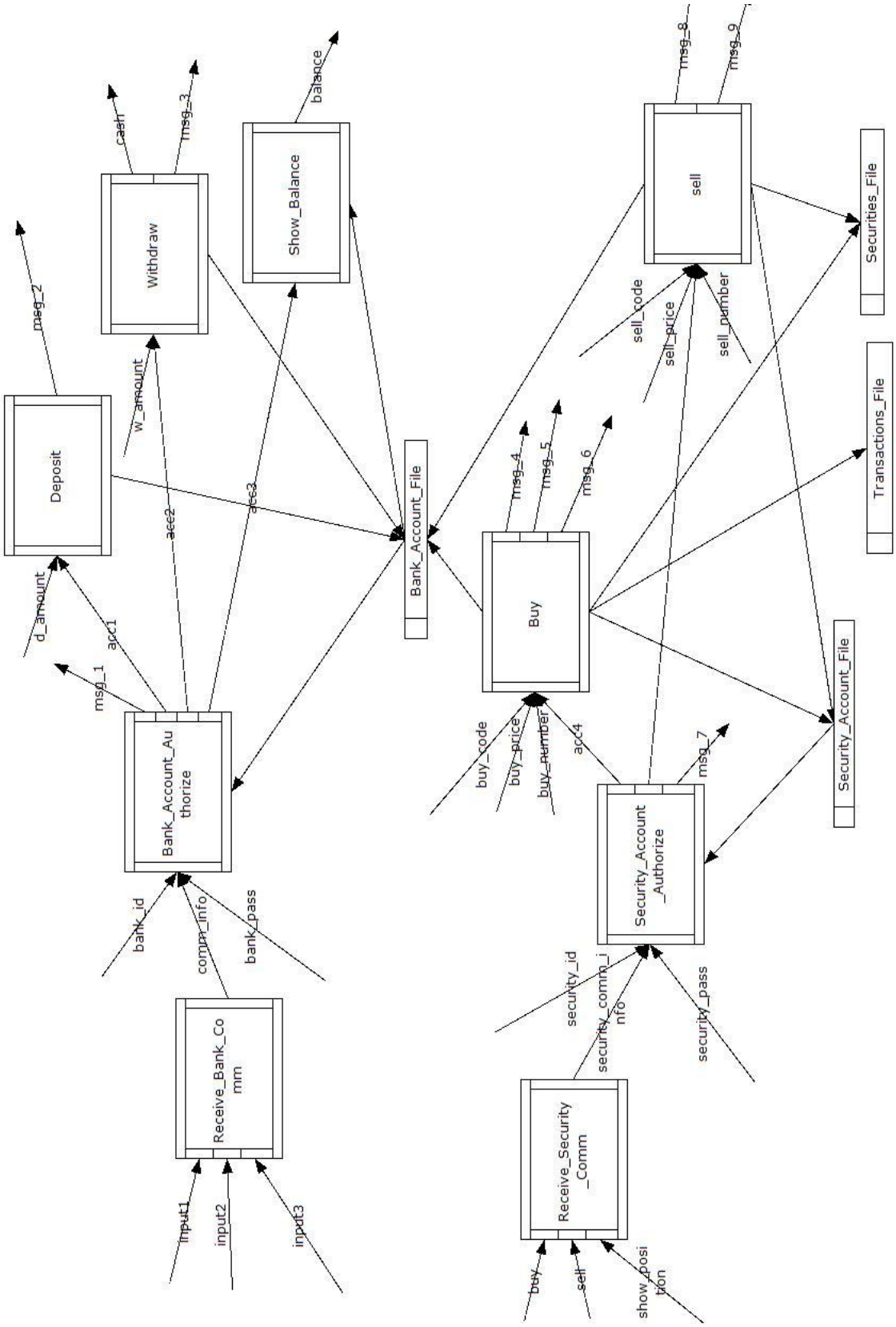
3.4 Maximum withdraw amount is 300,000. (F1.2.3)

3.5 ID number of each security account should be 10 digital numbers.(F1.4.2, F1.5.2, D2.2)

3.6 Password of each security account should be 6 digital numbers.(F1.4.2, F1.5.2, D2.2)

3.7 Security code should be 8 digital numbers.(F1.4.3, F1.5.3, D2.3)
3.8 Security code must be unique.(F1.4.3, F1.5.3, D2.3)
3.9 Transaction ID should be 20 digital numbers.(F1.4.7, F1.5.6, D2.4)
3.10 Target price in buy transaction can not less than the ask price of the security. (F1.4.5)
3.11 Target price in sell transaction can not higher than the bid price of the security. (F1.5.5)
3.12 The bid price of a security must less than the ask prices.(D2.3)

CDFD


Receive_Bank_Comm
input1
input2
input3
bank_id
comm_info
bank_pass
Bank_Account_Authorize
msg_1
d_amount
acc1
Deposit
msg_2
w_amount
acc2
Withdraw
cash
msg_3
acc3
Show_Balance
balance
Bank_Account_File
Receive_Security_Comm
buy
sell
show_position
security_id
security_comm_info
security_pass
Security_Account_Authorize
msg_7
acc4
buy_code
buy_price
buy_number
Buy
msg_4
msg_5
msg_6
sell_code
sell_price
sell_number
sell
msg_8
msg_9
Securities_File
Transactions_File
Security_Account_File

Module Specification

```
module SecurityTransactionSystem;


const
MAX_VALUE = 300000;

type
Bank_Account = composed of
                              id: nat0
                              name: string
                              password: nat0
                              balance: real
                             end;

Bank_Accounts: set of Bank_Account

Hold_Security = composed of
                             code: string
                             price: real
                             number: nat
                            end;

Hold_Securities: seq of Hold_Security;

Security_Account = composed of
                                 id: string
                                 name: string
                                 password: string
                                 hold_securities: Hold_Securities
                                end;

Security_Accounts: set of Security_Account

Comm_Info = {<deposite>, <withdraw>, <balance>, <buy>, <sell>, <position>}
Bank_Comm_Info = {<deposite>, <withdraw>, <balance>}
Security_Comm_Info = {<buy>, <sell>, <position>}

Action_Info = Bank_Comm_Info | Bank_Account
Transaction_info = Security_Comm_Info | Bank_Account

Security = composed of
                        code: string
                        bid: real
                        ask: real
                       end;

Securities = set of Security

Transaction = composed of
                          id: string
                          code: nat0
                          price: real
                          number: nat
                          type: Transaction_Type;
                          security_account: Security_Account
                         end;
```

```
var
ext #Bank_Account_File: Bank_Accounts
ext #Security_Account_File: Security_Accounts
ext #Securities_File: Securities


inv
forall[a: Bank_Account] | len(a.id) = 4;
forall[a: Bank_Account] | len(a.password) = 7;
forall[a, b: Bank_Account] | a <> b => a.id <> b.id;
forall[a: Security_Account] | len(a.id) = 10;
forall[a: Security_Account] | len(a.password) = 6;
forall[a, b: Security_Account] | a <> b => a.id <> b.id;
forall[a: Security] | len(a.code) = 10;
forall[a, b: Security] | a <> b => a.id <> b.id;
forall[a: Transaction] | len(a.id) = 20;
forall[a, b: Transaction] | a <> b => a.id <> b.id;


process Receive_Bank_Comm(input1: string | input2: string | input3: string)comm_info:
Security_Comm_Info
pre true
post input1 = "deposite" and comm_info = <deposite> or input2 = "withdraw" and comm_info =
<withdraw> or input3 = "show_balance" and comm_info = <show_balance>
end_process;

process Bank_Account_Authorize(comm_info: Security_Comm_Info, bank_id: string, bank_pass:
string)msg_1: string | acc1: Bank_Account | acc2: Account | acc3: Account
pre true
post (exists![x: Bank_Account_File] | (x.id = bank_id) and (bank_comm_info = <deposite> and
acc1 = x or bank_comm_info = <withdraw> and acc1 = x and bank_comm_info =
<show_balance> and acc3 = x)) or (exists![x: Bank_Account_File] | (x.id = bank_id) and msg_1 =
"Error!")
end_process;

process Deposit(acc1: Bank_Account, d_amount: real)msg_2: string
pre true
post acc1.balance = acc1.balance + deposite_amount or msg2 = "Deposite Success!"
end_process;

process Receive_Security_Comm(buy: string | sell: string | show_position:
string)security_comm_info: Comm_Info
pre true
post buy = "buy" and security_comm_info = <buy> or sell = "sell" and bank_comm_info = <sell>
or show_position = "show_position" and bank_comm_info = <show_position>
end_process;

process Security_Account_Authorize(security_comm_info: Security_Comm_Info, security_id:
string, security_pass: string)acc4: Security_Account | acc5 : Security_Account    | msg_7: string
ext    wr #Security_Account
pre true
post (exists![x: Security_Account_File] | (x.id = security_id) and (security_comm_info = <deposit>
and acc1 = x or security_comm_info = <sell> and acc5 = x or security_comm_info =
<show_position> and acc6 = x)) or ((exists![x: Security_Account_File] | (x.id = security_id)) and
```

```
msg_2 = "Error!")
end_process;

process Buy(acc4: Security_Account, buy_code: string, buy_price: real, buy_number: nat)msg_4:
string | msg_5: msg | msg_6:
ext rd #Bank_Account_File
        wr #Security_Account_File
        wr #Securities_File
pre true
post (exists![x: Security_Account_File and y: Security_Account_File and z: Securities_File] | (x.id
= acc4.id and y.security_account_id = acc4.id and z.code = buy_code and y.balance <= buy_
number * buy_ price and y.balance = y.balance - buy_number * buy_price and x.hold_securities =
union(acc4.hold_securities, { (buy_code, buy_price, buy_number)} and z.bid = buy_share_price
and msg_4 = "Success!") or (exists![x: Security_Account_File and y: Bank_Account_File and z:
Securities] | (x.id = acc1.id and y.security_account_id = acc4.id and z.code = buy_share_code
and y.balance < buy_number * buy_price and msg_5 = "Balance Is Not Enough!") or (exists![x:
Security_Account_File and y: Bank_Account_File and z: Securities_File] | (x.id = acc4.id and
y.security_account_id = acc4.id and z.code = buy_code and buy_prince < z.bid and msg_6 =
"The Offered Price Is Too Cheap!")
end_process;

process Withdraw(acc2: Bank_Account, w_amount: real)cash: real | msg_3: string
ext rd #Bank_Account_FIle
pre true
post acc2.balance <= withdraw_amount and acc2.balance = acc2.balacen + w_amount and cash
= w_amount or acc2.balance <= withdraw_amount and msg_2 = "Balance Is Not Enough!"
end_process;

process Show_Balance(acc3: Account)balance: real
ext rd #Bank_Account_FIle
pre    true
post    balance = acc3.balance
end_process;

process sell(acc5: Security_Account , sell_code: nat, sell_price: real, sell_number: real)msg_8:
msg | msg_9: msg
ext rd #Bank_Account_File
        wr #Security_Account_File
        wr #Securities_File
pre true
post (exists![x: Security_Account_File and y: Security_Account_File and z: Securities_File] | (x.id
= acc5.id and y.security_account_id = acc5.id and z.code = sell_code and sell_price <= z.ask and
y.balance -> y.balance - sell_number * sell_price) and x.hold_securities ->
diff(acc1.hold_securities, { (sell_share_code, sell_share_price, sell_share_number)} and z.ask =
sell_price) and msg_4 = "Success!") or (exists![x: Security_Account_File and y:
Security_Account_File and z: Securities_File] | (x.id = acc5.id and y.security_account_id = acc1.id
and z.code = sell_code and sell_prince > z.ask and msg_5 = "The Offered Price Is Too
Expensive!")
end_process;

end_module
```

Checklist for Checklist-based Inspection

Checklist

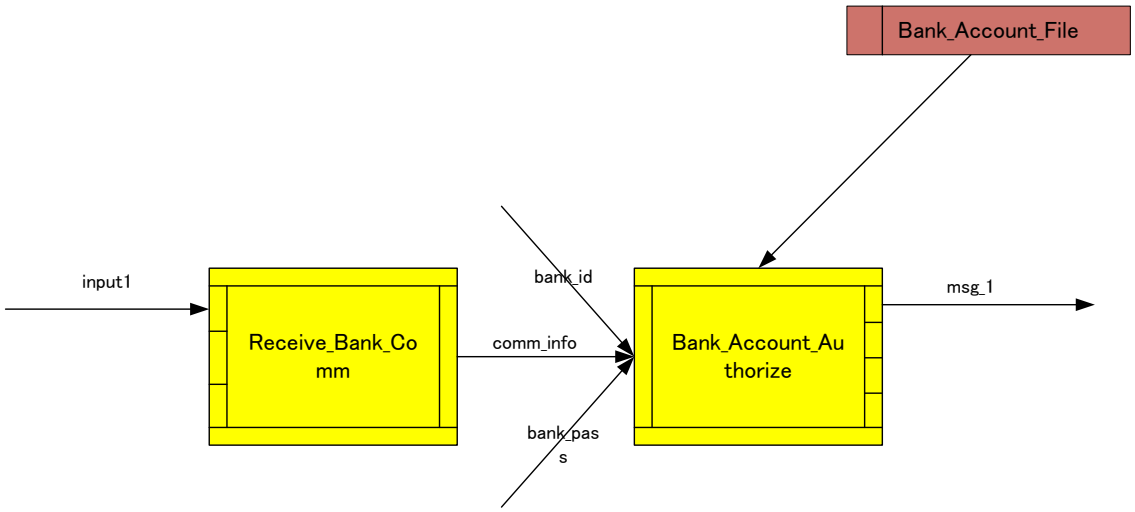
1. Whether function "Deposit" is implemented appropriately?
2. Whether function "Receive command" is implemented appropriately?
3. Whether function "Receive cash" is implemented appropriately?
4. Whether function "Update bank account balance" is implemented appropriately?
5. Whether function "Withdraw" is implemented appropriately?
6. Whether function "Check bank account password" is implemented appropriately?
7. Whether function "Receive withdraw amount" is implemented appropriately?
8. Whether function "Pay cash" is implemented appropriately?
9. Whether function "Check balance of bank account" is implemented appropriately?
10. Whether function "Show balance" is implemented appropriately?
11. Whether function "Buy securities" is implemented appropriately?
12. Whether function "Check securities and bank accounts password" is implemented appropriately?
13. Whether function "Receive security code and number" is implemented appropriately?
14. Whether function "Receive target price" is implemented appropriately?
15. Whether function "Check security price" is implemented appropriately?
16. Whether function "Check bank account balance" is implemented appropriately?
17. Whether function "Make and record transaction, update the balance" is implemented appropriately?
18. Whether function "Sell security" is implemented appropriately?
19. Whether data resource "Bank account database" is implemented appropriately?
20. Whether data resource "Security account database" is implemented appropriately?
21. Whether data resource "Securities database" is implemented appropriately?
22. Whether data resource "Transactions database" is implemented appropriately?
23. Whether constraint C3.1 about command is implemented appropriately?
24. Whether constraint C3.2 about ID of bank account is implemented appropriately?
25. Whether constraint C3.3 about Password of bank account is implemented appropriately?
26. Whether constraint C3.4 about maximum withdraw amount is implemented appropriately?
27. Whether constraint C3.5 about ID of security account is implemented appropriately?
28. Whether constraint C3.6 about Password of security account is implemented appropriately?

29. Whether constraint C3.7 about code of each security is implemented appropriately?
30. Whether constraint C3.8 about code of each security is implemented appropriately?
31. Whether constraint C3.9 about ID of transaction is implemented appropriately?
32. Whether constraint C3.10 about target prince in buy transaction is implemented appropriately?
33. Whether constraint C3.11 about target prince in sell transaction is implemented appropriately?
34. Whether constraint C3.12 about bid and ask prices is implemented appropriately?

Checklist for Proposed Inspection Approach

Questions for system scenario 01

{input1}[Receive_Bank_Comm, Bank_Account_Authorize]{msg_1}



1. Questions for process "Receive_Bank_Comm"

process Receive_Bank_Comm(input1: **string**) comm_info: **Security_Comm_info**
pre true
post input1 = "deposit" and comm_info = <deposit>
end_process

Q1: Whether the name "**Receive_Bank_Comm**" is appropriate?

Q2: Whether the name and type of input variable "**input1**" are appropriate?

Q3: Whether the name and type of output variable "**comm_info**" are appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
input1	string
comm_info	Security_Comm_Info

Q8: Whether the invariants related to process "**Receive_Bank_Comm**" are defined correctly?

Q9: What constraints in the informal specification are formalized by these invariant?

2. Questions for process “Bank_Account_Authorize”

```
process Bank_Account_Authorize(bank_id: string, bank_pass: string, comm_info:
Security_Comm_info) msg_1: string
pre true
post exists![x: Bank_Account_File] | (x.id = bank_id) and msg_1 = "Error!"
end_process
```

Q1: Whether the name “**Bank_Account_Authorize**” is appropriate?

Q2: Whether the name and type of input variables “**bank_id**”, “**bank_pass**”, and “**comm_info**” are appropriate?

Q3: Whether the name and type of output variable “**msg_1**” is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
Bank_Account_File	Security_Accounts
x.id	nat0
bank_id	string

Q8: Whether the data store “**Bank_Account_File**” is declared appropriately with “**wr**” or “**rd**”?

Q9: What data resources in the formal specification is formalized by the data store “**Bank_Account_File**”?

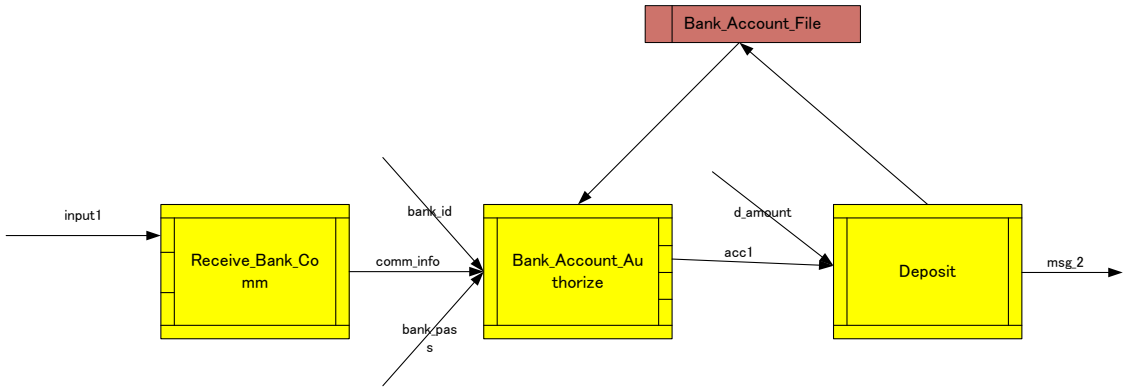
Q10: Whether the invariants related to process “**Bank_Account_Authorize**” are defined correctly?

```
forall[a: Bank_Account] | len(a.id) = 4;
forall[a: Bank_Account] | len(a.password) = 7;
forall[a, b: Bank_Account] | a <> b => a.id <> b.id;
```

Q11: What constraints in the informal specification are formalized by these invariant?

Questions for system scenario 02

{input1}[Receive_Bank_Comm, Bank_Account_Authorize, Deposit]{msg_2}



1. Questions for process "Bank_Account_Authorize"

process Bank_Account_Authorize (bank_id: **string**, bank_pass: **string**, comm_info: **Security_Comm_info**) acc1: **Bank_Account**
pre true
post (exists![x: **Bank_Account_File**] | (x.id = bank_id) and (bank_comm_info = <deposite>))
and acc1 = x
end_process

Q1: Whether the name "**Bank_Account_Authorize**" is appropriate?

Q2: Whether the name and type of input variables "**bank_id**", "**bank_pass**", and "**comm_info**" are appropriate?

Q3: Whether the name and type of output variable "**acc1**" is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
bank_comm_info	undefined
x.id	nat0
bank_id	string
acc1	Bank_Account

Q8: Whether the data store "**Bank_Account_File**" is declared appropriately with "**wr**" or "**rd**"?

Q9: What data resources in the formal specification is formalized by the data store "**Bank_Account_File**"?

Q10: Whether the invariants related to process "**Bank_Account_Authorize**" are defined correctly?

```
forall[a: Bank_Account] | len(a.id) = 4;
forall[a: Bank_Account] | len(a.password) = 7;
forall[a, b: Bank_Account] | a <> b => a.id <> b.id;
```

Q11: What constraints in the informal specification are formalized by these invariant?

2. Questions for process "Deposit"

```
process Deposit (acc1: Bank_Account,  d_amount: real) msg_2: string
pre true
post acc1.balance = acc1.balance + deposite_amount
end_process
```

Q1: Whether the name "**Deposit**" is appropriate?

Q2: Whether the name and type of input variables "**acc1**", and "**d_amount**" are appropriate?

Q3: Whether the name and type of output variable "**msg_2**" are appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
acc1.balance	real
deposite_amount	undefined

Q8: Whether the data store "**Bank_Account_File**" is declared appropriately with "**wr**" or "**rd**"?

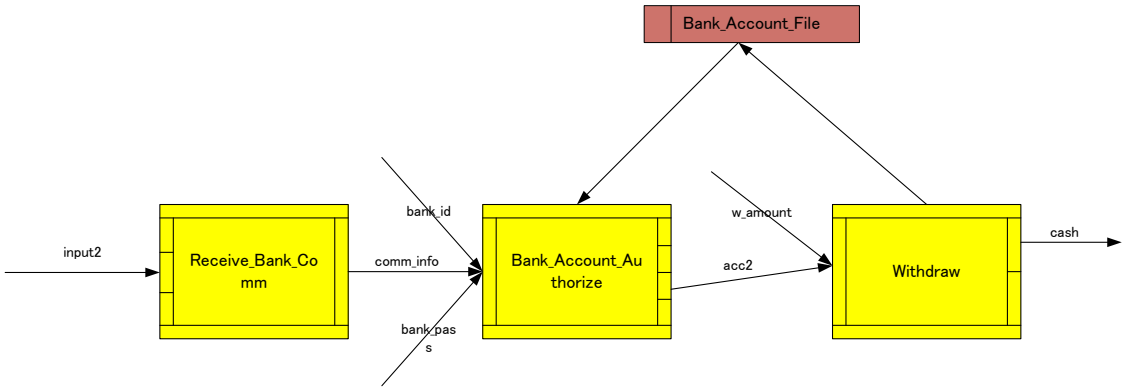
Q9: Whether the invariants related to process "**Bank_Account_Authorize**" are defined correctly?

```
forall[a: Bank_Account] | len(a.id) = 4;
forall[a: Bank_Account] | len(a.password) = 7;
forall[a, b: Bank_Account] | a <> b => a.id <> b.id;
```

Q10: What constraints in the informal specification are formalized by these invariant?

Questions for system scenario 03

{input2}[Receive_Bank_Comm, Bank_Account_Authorize, Withdraw]{cash}



1. Questions for process “Receive_Bank_Comm”

process Receive_Bank_Comm(input2: **string**) comm_info: **Security_Comm_info**
pre true
post input1 = “withdraw” and comm_info = <withdraw>
end_process

Q1: Whether the name “**Receive_Bank_Comm**” is appropriate?

Q2: Whether the name and type of input variable “**input2**” are appropriate?

Q3: Whether the name and type of output variable “**comm_info**” are appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
input2	string
comm_info	Security_Comm_Info

Q8: Whether the invariants related to process "**Receive_Bank_Comm**" are defined correctly?

Q9: What constraints in the informal specification are formalized by these invariant?

2. Questions for process "Bank_Account_Authorize"

process Bank_Account_Authorize (bank_id: **string**, bank_pass: **string**, comm_info: **Security_Comm_info**) acc2: **Account**
pre true
post exists![x: **Bank_Account_File**] | bank_comm_info = <withdraw> **and** acc1 = x
and
bank_comm_info = <show_balance> **and** acc3 = x
end_process

Q1: Whether the name "**Bank_Account_Authorize**" is appropriate?

Q2: Whether the name and type of input variables "**bank_id**", "**bank_pass**", and "**comm_info**" are appropriate?

Q3: Whether the name and type of output variable "**acc2**" is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
bank_comm_info	undefined
x.id	nat0
acc1	undefined
acc3	undefined

Q8: Whether the data store "**Bank_Account_File**" is declared appropriately with "**wr**" or "**rd**"?

Q9: What data resources in the formal specification is formalized by the data store "**Bank_Account_File**"?

Q10: Whether the invariants related to process "**Bank_Account_Authorize**" are defined correctly?

forall[a: Bank_Account] | len(a.id) = 4;
forall[a: Bank_Account] | len(a.password) = 7;
forall[a, b: Bank_Account] | a <> b => a.id <> b.id;

Q11: What constraints in the informal specification are formalized by these invariant?

3. Questions for process "Withdraw"

```
process Withdraw (acc2: Bank_Account, w_amount: real) cash: real
ext rd #Bank_Account_File
pre true
post acc2.balance <= withdraw_amount and
     acc2.balance = acc2.balacen + w_amount and
     cash = w_amount
end_process
```

Q1: Whether the name "**Withdraw**" is appropriate?

Q2: Whether the name and type of input variables "**acc2**", and "**w_amount**" are appropriate?

Q3: Whether the name and type of output variable "**cash**" is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
acc2.balance	real
withdraw_amount	undefined
w_amount	real

Q8: Whether the data store "**Bank_Account_File**" is declared appropriately with "**wr**" or "**rd**"?

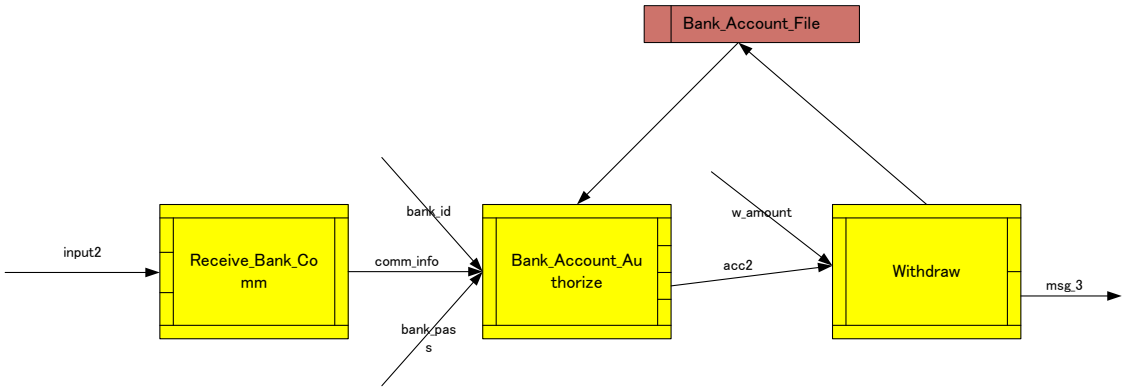
Q9: Whether the invariants related to process "**Bank_Account_Authorize**" are defined correctly?

```
forall[a: Bank_Account] | len(a.id) = 4;
forall[a: Bank_Account] | len(a.password) = 7;
forall[a, b: Bank_Account] | a <> b => a.id <> b.id;
```

Q10: What constraints in the informal specification are formalized by these invariant?

Questions for system scenario 04

{input2}[Receive_Bank_Comm, Bank_Account_Authorize, Withdraw]{msg_3}



1. Questions for process "Withdraw"

process Withdraw (acc2: **Bank_Account**, w_amount: **real**) msg_3: **string**
ext rd #Bank_Account_File
pre true
post acc2.balance <= withdraw_amount **and** msg_2 = "Balance Is Not Enough!"
end_process

Q1: Whether the name "**Withdraw**" is appropriate?

Q2: Whether the name and type of input variables "**acc2**", and "**w_amount**" are appropriate?

Q3: Whether the name and type of output variable "**msg_3**" is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
acc2.balance	real
msg_2	undefined

Q8: Whether the data store "**Bank_Account_File**" is declared appropriately with "**wr**" or "**rd**"?

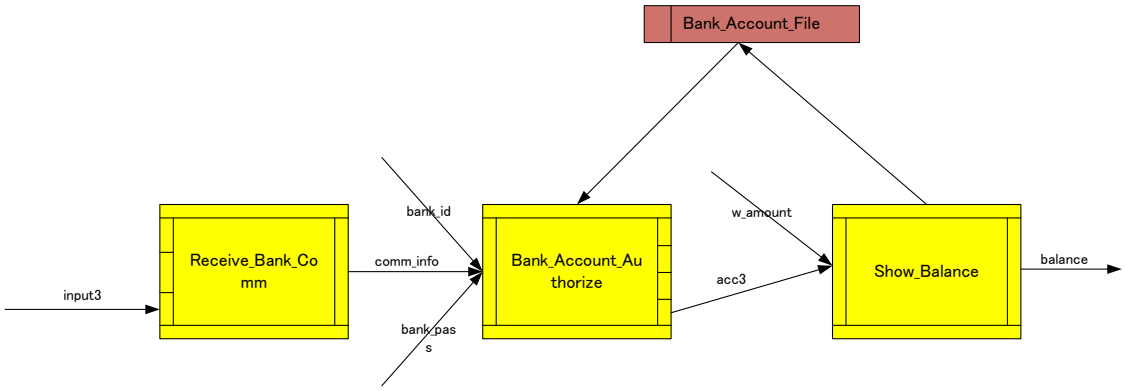
Q9: Whether the invariants related to process "**Bank_Account_Authorize**" are defined correctly?

forall[a: Bank_Account] | len(a.id) = 4;
forall[a: Bank_Account] | len(a.password) = 7;
forall[a, b: Bank_Account] | a <> b => a.id <> b.id;

Q10: What constraints in the informal specification are formalized by these invariant?

Questions for system scenario 05

{input3}[Receive_Bank_Comm, Bank_Account_Authorize, Show_Balance]{balance}



1. Questions for process “Receive_Bank_Comm”

process Receive_Bank_Comm(input3: **string**) comm_info: **Security_Comm_info**
pre true
post input3 = "show_balance" **and** comm_info = <show_balance>
end_process

Q1: Whether the name “**Receive_Bank_Comm**” is appropriate?

Q2: Whether the name and type of input variable “**input3**” are appropriate?

Q3: Whether the name and type of output variable “**comm_info**” are appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
input3	string
comm_info	Security_Comm_Info

Q8: Whether the invariants related to process "**Receive_Bank_Comm**" are defined correctly?

Q9: What constraints in the informal specification are formalized by these invariant?

2. Questions for process "Bank_Account_Authorize"

process Bank_Account_Authorize (bank_id: **string**, bank_pass: **string**, comm_info: **Security_Comm_info**) acc3: **Account**
pre true
post exists![x: **Bank_Account_File**] | bank_comm_info = <withdraw> **and** acc1 = x
and
bank_comm_info = <show_balance> **and** acc3 = x
end_process

Q1: Whether the name "**Bank_Account_Authorize**" is appropriate?

Q2: Whether the name and type of input variables "**bank_id**", "**bank_pass**", and "**comm_info**" are appropriate?

Q3: Whether the name and type of output variable "**acc3**" is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
bank_comm_info	undefined
x.id	nat0
acc1	undefined
acc3	Account

Q8: Whether the data store "**Bank_Account_File**" is declared appropriately with "**wr**" or "**rd**"?

Q9: What data resources in the formal specification is formalized by the data store "**Bank_Account_File**"?

Q10: Whether the invariants related to process "**Bank_Account_Authorize**" are defined correctly?

forall[a: Bank_Account] | len(a.id) = 4;
forall[a: Bank_Account] | len(a.password) = 7;
forall[a, b: Bank_Account] | a <> b => a.id <> b.id;

Q11: What constraints in the informal specification are formalized by these invariant?

3. Questions for process “Show_Balance”

```
process Show_Balance (acc3: Account) balance: real
ext rd #Bank_Account_File
pre true
post balance = acc3.balance
end_process
```

Q1: Whether the name “**Show_Balance**” is appropriate?

Q2: Whether the name and type of input variables “**acc3**” are appropriate?

Q3: Whether the name and type of output variable “**balance**” is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

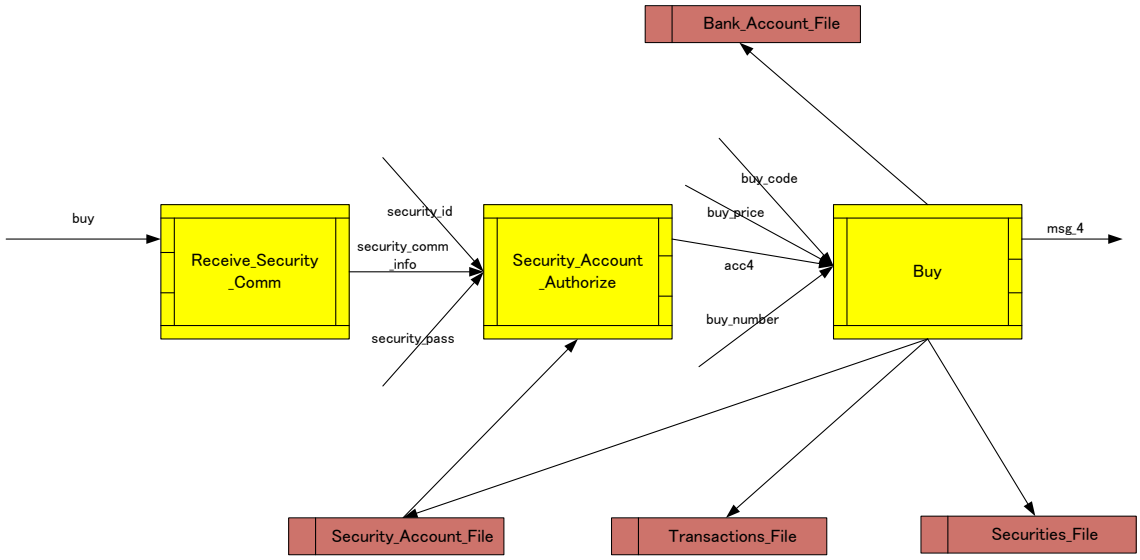
Q8: Whether the data store “**Bank_Account_File**” is declared appropriately with “**wr**” or “**rd**”?

Q9: Whether the invariants related to process “**Bank_Account_Authorize**” are defined correctly?

Q10: What constraints in the informal specification are formalized by these invariant?

Questions for system scenario 06

{buy}[Receive_Security_Comm, Security_Account_Authorize, Buy]{msg_4}



1. Questions for process "Receive_Security_Comm"

process Receive_Security_Comm (buy: **string**) security_comm_info: **Comm_Info**
pre true
post buy = "buy" and security_comm_info = <buy>
end_process

Q1: Whether the name "**Receive_Security_Comm**" is appropriate?

Q2: Whether the name and type of input variable "**buy**" are appropriate?

Q3: Whether the name and type of output variable "**security_comm_info**" are appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Q8: Whether the invariants related to process "**Receive_Security_Comm**" are defined correctly?

Q9: What constraints in the informal specification are formalized by these invariant?

2. Questions for process "Security_Account_Authorize"

process Security_Account_Authorize (security_comm_info: **Security_Comm_Info**, security_id: **string**, security_pass: **string**)acc4: **Security_Account**
ext wr #Security_Account
pre true
post exists![x: Security_Account_File] | (x.id = security_id) and security_comm_info = <deposit> and acc1 = x
end_process

Q1: Whether the name "**Security_Account_Authorize**" is appropriate?

Q2: Whether the name and type of input variables "**security_comm_info**", "**security_id**", and "**security_pass**" are appropriate?

Q3: Whether the name and type of output variable "**acc4**" is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
Security_Account_File	Security_Accounts
x.id	string
acc1	undefined

Q8: Whether the data store "**Security_Account_File**" is declared appropriately with "**wr**" or "**rd**"?

Q9: What data resources in the formal specification is formalized by the data store "**Security _Account_File**"?

Q10: Whether the invariants related to process "**Security_Account_Authorize**" are defined correctly?

forall[a: Security_Account] | len(a.id) = 10;
forall[a: Security_Account] | len(a.password) = 6;
forall[a, b: Security_Account] | a <> b => a.id <> b.id;

Q11: What constraints in the informal specification are formalized by these invariant?

3. Questions for process “Buy”

```
process Buy (acc4: Security_Account, buy_code: string, buy_price: real, buy_number: nat)
                                                                          msg_4: string
ext rd #Bank_Account_File
    wr #Security_Account_File
    wr #Securities_File
pre true
post exists![x: Security_Account_File and y: Security_Account_File and z: Securities_File] |
          (x.id = acc4.id and y.security_account_id = acc4.id and z.code = buy_code and
            y.balance <= buy_ number * buy_ price and
            y.balance = y.balance - buy_number * buy_price and
            x.hold_securities = union(acc4.hold_securities,
                                      { (buy_code, buy_price, buy_number)} and
            z.bid = buy_share_price and
            msg_4 = "Success!")
end_process
```

Q1: Whether the name “**Buy**” is appropriate?

Q2: Whether the name and type of input variables “**acc4**”, "**buy_code**", "**buy_price**", "**buy_number**" are appropriate?

Q3: Whether the name and type of output variable “**msg_4**” is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
Security_Account_File	Security_Accounts
y.security_account_id	undefined
z.code	string

Q8: Whether the data store “**Security_Account_File**”, “**Bank_Account_File**”, “**Securities_File**”, “**Transaction_File**” are declared appropriately with “**wr**” or “**rd**”?

Q9: What data resources in the formal specification is formalized by the data store “**Security_Account_File**”, “**Securities_File**”?

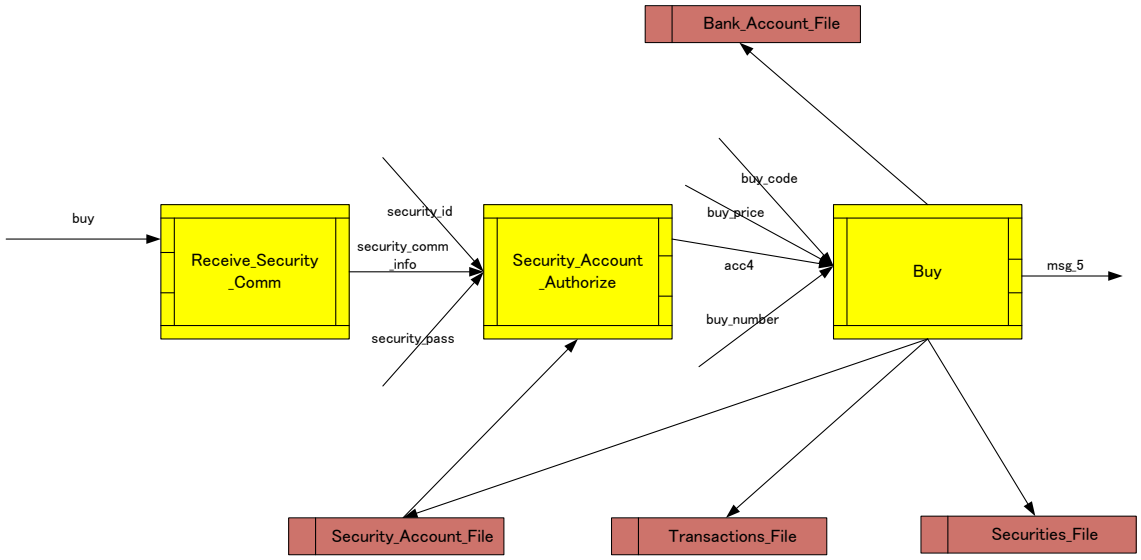
Q10: Whether the invariants related to process “**Buy**” are defined correctly

```
forall[a: Security_Account] | len(a.id) = 10;
forall[a: Security_Account] | len(a.password) = 6;
forall[a, b: Security_Account] | a <> b => a.id <> b.id;
forall[a: Security] | len(a.code) = 10;
forall[a, b: Security] | a <> b => a.id <> b.id;
```

Q11: What constraints in the informal specification are formalized by these invariant?

Questions for system scenario 07

{buy}[Receive_Security_Comm, Security_Account_Authorize, Buy]{msg_5}



1. Questions for process “Buy”

process Buy (acc4: **Security_Account**, buy_code: **string**, buy_price: **real**, buy_number: **nat**)
msg_5: **string**

ext rd #Bank_Account_File
wr #Security_Account_File
wr #Securities_File
pre true
post exists![x: Security_Account_File and y: Bank_Account_File and z: Securities] |
(x.id = acc1.id and y.security_account_id = acc4.id and z.code = buy_share_code
and
y.balance < buy_number * buy_price and
msg_5 = "Balance Is Not Enough!")
end_process

Q1: Whether the name “**Buy**” is appropriate?

Q2: Whether the name and type of input variables “**acc4**”, "**buy_code**", "**buy_price**", "**buy_number**" are appropriate?

Q3: Whether the name and type of output variable “**msg_5**” is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
Security_Account_File	Security_Accounts
Bank_Account_File	Bank_Accounts
Securities_File	Securities
x.id	string
acc4.id	string
y.security_account_id	undefined
z.code	string

Q8: Whether the data store "**Security_Account_File**", "**Bank_Account_File**", "**Securities_File**", "**Transaction_File**" are declared appropriately with "**wr**" or "**rd**"?

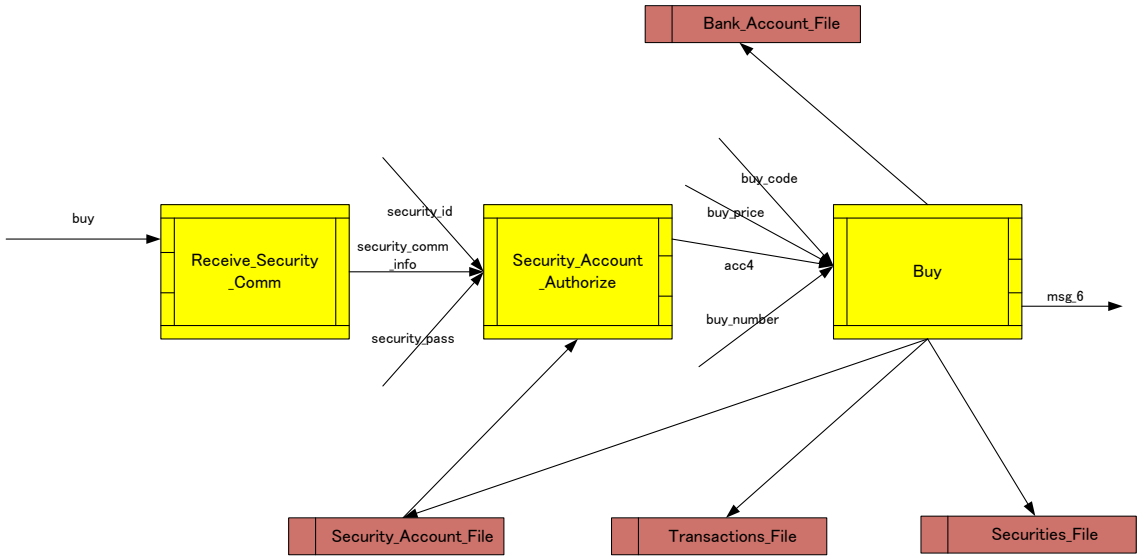
Q9: What data resources in the formal specification is formalized by the data store "**Security_Account_File**", "**Bank_Account_File**", "**Securities_File**"?

Q10: Whether the invariants related to process "**Buy**" are defined correctly

Q11: What constraints in the informal specification are formalized by these invariant?

Questions for system scenario 08

{buy}[Receive_Security_Comm, Security_Account_Authorize, Buy]{msg_6}



1. Questions for process “Buy”

process Buy (acc4: **Security_Account**, buy_code: **string**, buy_price: **real**, buy_number: **nat**)
msg_6:

ext rd #Bank_Account_File
wr #Security_Account_File
wr #Securities_File
pre true
post exists![x: Security_Account_File and y: Bank_Account_File and z: Securities_File] |
(x.id = acc4.id and y.security_account_id = acc4.id and z.code = buy_code
and
buy_prince < z.bid and
msg_6 = "The Offered Price Is Too Cheap!")
end_process

Q1: Whether the name “**Buy**” is appropriate?

Q2: Whether the name and type of input variables “**acc4**”, "**buy_code**", "**buy_price**", "**buy_number**" are appropriate?

Q3: Whether the name and type of output variable “**msg_6**” is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Variable	Type
Security_Account_File	Security_Accounts
Bank_Account_File	Bank_Accounts
Securities_File	Securities
x.id	string
acc4.id	string
y.security_account_id	undefined
z.code	string

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

```
forall[a: Bank_Account] | len(a.id) = 4;
forall[a: Bank_Account] | len(a.password) = 7;
forall[a, b: Bank_Account] | a <> b => a.id <> b.id;
forall[a: Security_Account] | len(a.id) = 10;
forall[a: Security_Account] | len(a.password) = 6;
forall[a, b: Security_Account] | a <> b => a.id <> b.id;
forall[a: Security] | len(a.code) = 10;
forall[a, b: Security] | a <> b => a.id <> b.id;
```

Q8: Whether the data store "**Security_Account_File**", "**Bank_Account_File**", "**Securities_File**", "**Transaction_File**" are declared appropriately with "**wr**" or "**rd**"?

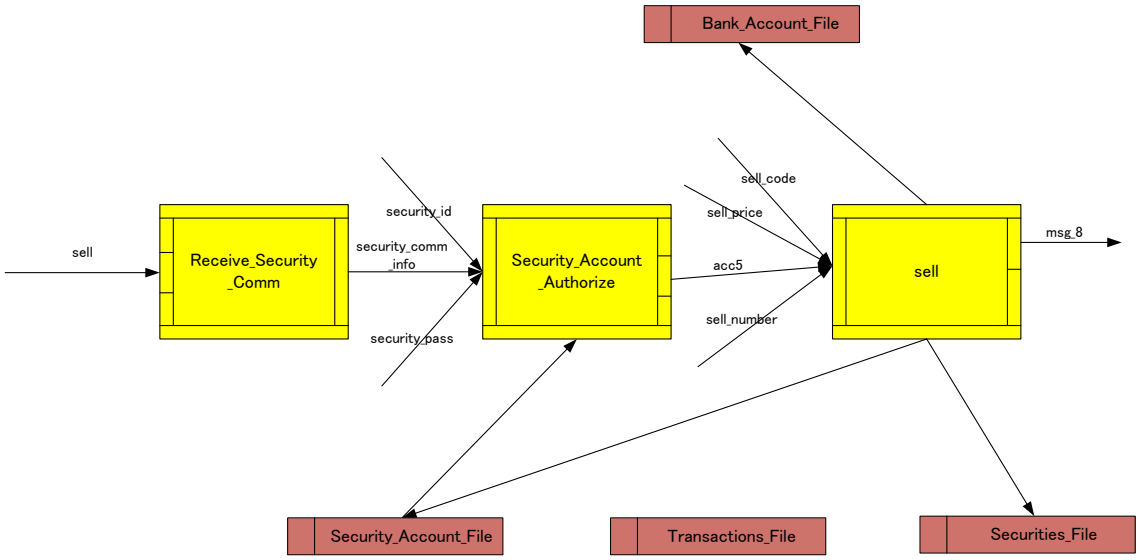
Q9: What data resources in the formal specification is formalized by the data store "**Security_Account_File**", "**Bank_Account_File**", "**Securities_File**"?

Q10: Whether the invariants related to process "**Buy**" are defined correctly

Q11: What constraints in the informal specification are formalized by these invariant?

Questions for system scenario 09

{sell}[Receive_Security_Comm, Security_Account_Authorize, sell]{msg_8}



1. Questions for process “Receive_Security_Comm”

process Receive_Security_Comm (sell: **string**) security_comm_info: **Comm_Info**
pre true
post sell = "sell" and security_comm_info = <sell>
end_process

Q1: Whether the name “**Receive_Security_Comm**” is appropriate?

Q2: Whether the name and type of input variable “**sell**” are appropriate?

Q3: Whether the name and type of output variable “**security_comm_info**” are appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Q8: Whether the invariants related to process “**Receive_Security_Comm**” are defined correctly?

Q9: What constraints in the informal specification are formalized by these invariant?

2. Questions for process “Security_Account_Authorize”

```
process Security_Account_Authorize (security_comm_info: Security_Comm_Info, security_id:
string, security_pass: string)acc5: Security_Account
ext wr #Security_Account
pre true
post exists![x: Security_Account_File] | (x.id = security_id) and
                security_comm_info = <sell> and acc5 = x
end_process
```

Q1: Whether the name “**Security_Account_Authorize**” is appropriate?

Q2: Whether the name and type of input variables “**security_comm_info**”, “**security_id**”, and “**security_pass**” are appropriate?

Q3: Whether the name and type of output variable “**acc5**” is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
Security_Account_File	Security_Accounts
x.id	string
acc5	Security_Account

Q8: Whether the data store “**Security_Account_File**” is declared appropriately with “**wr**” or “**rd**”?

Q9: What data resources in the formal specification is formalized by the data store “**Security_Account_File**”?

Q10: Whether the invariants related to process “**Security_Account_Authorize**” are defined correctly?

```
forall[a: Security_Account] | len(a.id) = 10;
forall[a: Security_Account] | len(a.password) = 6;
forall[a, b: Security_Account] | a <> b => a.id <> b.id;
```

Q11: What constraints in the informal specification are formalized by these invariant?

3. Questions for process “sell”

```
process sell(acc5: Security_Account , sell_code: nat, sell_price: real, sell_number: real)
                                                                    msg_8: msg
ext rd #Bank_Account_File
    wr #Security_Account_File
    wr #Securities_File
pre true
post exists![x: Security_Account_File and y: Security_Account_File and z: Securities_File] |
            (x.id = acc5.id and y.security_account_id = acc5.id and z.code = sell_code and
             sell_price <= z.ask and
             y.balance -> y.balance - sell_number * sell_price) and
             x.hold_securities -> diff(acc1.hold_securities,
                                 { (sell_share_code, sell_share_price, sell_share_number)} and
             z.ask = sell_price and msg_4 = "Success!"
end_process
```

Q1: Whether the name “**sell**” is appropriate?

Q2: Whether the name and type of input variables “**acc5**”, "**sell_code**", "**sell_price**", "**sell_number**" are appropriate?

Q3: Whether the name and type of output variable “**msg_8**” is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
x.id	string
acc5.id	string
y.security_account_id	undefined
msg_4	undefined

Q8: Whether the data store “**Security_Account_File**”, “**Bank_Account_File**”, “**Securities_File**”, “**Transaction_File**” are declared appropriately with “**wr**” or “**rd**”?

Q9: What data resources in the formal specification is formalized by the data store “**Security_Account_File**”, “**Securities_File**”?

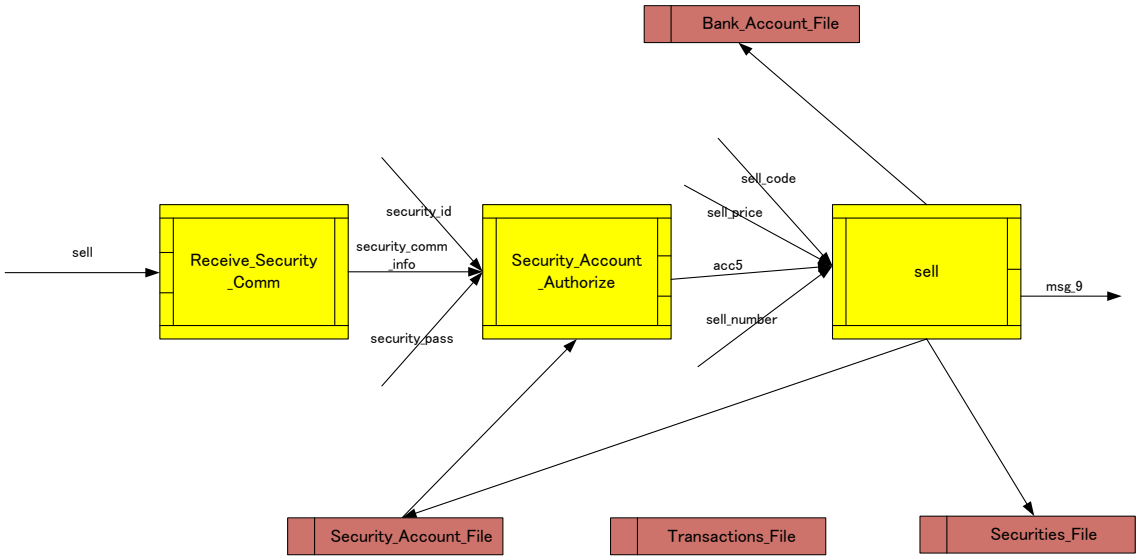
Q10: Whether the invariants related to process “**sell**” are defined correctly

```
forall[a: Security_Account] | len(a.id) = 10;
forall[a: Security_Account] | len(a.password) = 6;
forall[a, b: Security_Account] | a <> b => a.id <> b.id;
forall[a: Security] | len(a.code) = 10;
forall[a, b: Security] | a <> b => a.id <> b.id;
```

Q11: What constraints in the informal specification are formalized by these invariant?

Questions for system scenario 10

{sell}[Receive_Security_Comm, Security_Account_Authorize, sell]{msg_9}



1. Questions for process “sell”

process **sell**(acc5: **Security_Account** , sell_code: **nat**, sell_price: **real**, sell_number: **real**)
msg_9: **msg**
ext rd #Bank_Account_File
wr #Security_Account_File
wr #Securities_File
pre true
post exists![x: Security_Account_File and y: Security_Account_File and z: Securities_File] |
(x.id = acc5.id and y.security_account_id = acc1.id and z.code = sell_code
and
sell_prince > z.ask and
msg_5 = "The Offered Price Is Too Expensive!")
end_process

Q1: Whether the name “**sell**” is appropriate?

Q2: Whether the name and type of input variables “**acc5**”, "**sell_code**", "**sell_price**", "**sell_number**" are appropriate?

Q3: Whether the name and type of output variable “**msg_8**” is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
Security_Account_File	Security_Accounts
Securities_File	Securities
x.id	string
acc5.id	string
y.security_account_id	undefined
msg_5	undefined

Q8: Whether the data store "**Security_Account_File**", "**Bank_Account_File**", "**Securities_File**", "**Transaction_File**" are declared appropriately with "**wr**" or "**rd**"?

Q9: What data resources in the formal specification is formalized by the data store "**Security_Account_File**", "**Securities_File**"?

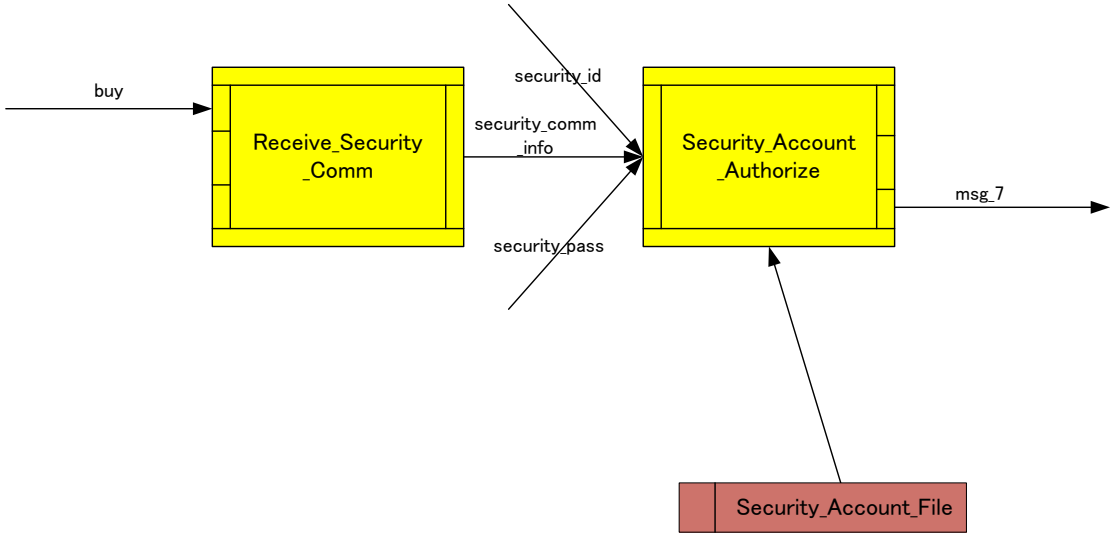
Q10: Whether the invariants related to process "**sell**" are defined correctly

```
forall[a: Security_Account] | len(a.id) = 10;
forall[a: Security_Account] | len(a.password) = 6;
forall[a, b: Security_Account] | a <> b => a.id <> b.id;
forall[a: Security] | len(a.code) = 10;
forall[a, b: Security] | a <> b => a.id <> b.id;
```

Q11: What constraints in the informal specification are formalized by these invariant?

Questions for system scenario 11

{buy}[Receive_Security_Comm, Security_Account_Authorize]{msg_7}



1. Questions for process “Security_Account_Authorize”

process Security_Account_Authorize (security_comm_info: **Security_Comm_Info**, security_id: **string**, security_pass: **string**)msg_7: **String**
ext wr #Security_Account
pre true
post exists![x: Security_Account_File] | (x.id = security_id) and msg_2 = "Error!"
end_process

Q1: Whether the name “**Security_Account_Authorize**” is appropriate?

Q2: Whether the name and type of input variables “**security_comm_info**”, “**security_id**”, and “**security_pass**” are appropriate?

Q3: Whether the name and type of output variable “**acc5**” is appropriate?

Q4: What functions in the informal specification are formalized by this process?

Q5: What constraints in the informal specification are formalized by this process?

Q6: Whether all input and output variables are used in the pre- and post-condition?

Q7: Whether the variables and their types in the pre- and post-condition are used appropriately?

Variable	Type
Security_Account_File	Security_Accounts
msg_2	undefined

Q8: Whether the data store "**Security_Account_File**" is declared appropriately with "**wr**" or "**rd**"?

Q9: What data resources in the formal specification is formalized by the data store "**Security _Account_File**"?

Q10: Whether the invariants related to process "**Security_Account_Authorize**" are defined correctly?

```
forall[a: Security_Account] | len(a.id) = 10;
forall[a: Security_Account] | len(a.password) = 6;
forall[a, b: Security_Account] | a <> b => a.id <> b.id;
```

Q11: What constraints in the informal specification are formalized by these invariant?