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

LI, Mo / 李, 漠

(出版者 / Publisher)
法政大学大学院情報科学研究科

(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編

(巻 / Volume)
11

(開始ページ / Start Page)
1

(終了ページ / End Page)
13

(発行年 / Year)
2015-09-15

博士学位論文
論文内容の要旨および審査結果の要旨

論文題目	A Systematic Inspection Approach to Verifying and Validating Formal Specifications based on Specification Animation and Traceability
氏　　名	李　漢
学位の種類	博士（理学）
学位授与年月日	2015 年 9 月 15 日
学位授与の条件	法政大学学位規則第 5 条第 1 項第 1 号該当者（甲）
論文審査委員	主　　査　　小池　誠彦　　教授
	副　　査　　劉　　少英　　教授
	副　　査　　雪田　修一　　教授
	副　　査　　藤田　悟　　　教授

2015年 6月 12日

学位論文審査委員会

委員長　　花泉 弘　　　殿

学位論文審査小委員会

主査　教授　小池　誠彦　㊞

副査　教授　　劉　少英　㊞ 劉

副査　教授　雪田　修一　㊞

副査　教授　藤田　悟　藤田

試問による学識確認の報告

法政大学学位規則第１９条により、　　李 漠（Mo Li）　　氏について、その論文を中心に関連する学問領域の試問を行った結果、合格と判定した。

以　上

（報告様式Ｉ）

2015 年　6 月　12 日

学位論文審査委員会

委員長　　花泉　弘　　殿

学位論文審査小委員会

主査　教授　　小池　誠彦

副査　教授　　劉　少英

副査　教授　　雪田　修一

副査　教授　　藤田　悟

李　漠　(Mo Li)　氏　提出学位請求論文

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

論文内容の要旨と審査結果の要旨（報告）

（報告様式Ⅱ）

１．論文内容の要旨

本論文はソフトウエアのフォーマルな要求定義記述を検証し妥当性チェックを行うため、その前工程とも言える構造化された自然言語で書かれたインフォーマルな要求記述と照らし合わせながら、フォーマル記述の不備、不整合、欠けなどを発見するためのシステム的なインスペクション方式を提案している。フォーマルな要求定義記述を実際に“形式的に実行”することで、とりうる全ての振舞いをアニメーション（シナリオ）として抽出し利用者に提示することで仕様の誤りや欠けを発見することを可能とした。さらにフォーマル要求仕様記述とインフォーマル要求仕様記述の間の対応関係をシステムが把握することができるので、未チェックな要求があればチェックリストとしてシステムが提示し利用者が答える形で、妥当性のあるフォーマル要求定義記述を得ることを可能とするソフトウエアシステム（フレームワーク）を開発している。

システム的なインスペクションとは、構造化された自然言語で書かれたインフォーマル記述とコンディション・データーフロー・ダイヤグラム（CDFD: Condition Data Flow Diagram）で表現されるフォーマル記述とを突き合わせることで、インフォーマル/フォーマル記述間を追跡し対応づけが可能であることに着目し、仕様記述の不備、不整合などをシステムが指摘するものである。利用者と対話形式に追跡を行いながら、インフォーマル記述とフォーマル記述の整合をとって行く。システムが不整合や未確認な記述を検出すると利用者に質問として対応を求め、その結果が記述に反映されていく。

得られた、フォーマル記述は、CDFD のネットワークに表現されているので形式的に実行が可能となる。この CDFD を流れる実行可能な個々の流れがそれぞれシナリオに対応することになる。本論分ではこのシナリオをアニメーションと呼ぶ。入出力データは形式的（数学的に正確）に記述されているので、実際に実データ群を流す必要が無く効率が良い。従い全てのアニメーションを洗い出すことが可能になる。利用者はそのアニメーションの結果を履歴として GUI で確認できるので未検証なシナリオが容易に発見できる利点がある。CDFD 記述は階層化を前提としているので、それぞれの階層で適切なモジュール（プロセス）の数の繋がりとして切り出して解析できるので提案システムは実用性の面でも規模が大きくなっても有効であると言える。

さらに上記機能を組み込んだ対話的なインスペクション・システムのプロトタイプを構築している。コードサイズはＣ＃で数万行にも及び、SOFL ツール支援システムとして学生の利用に供されている。このフレームワーク上で、複数の仕様バグが混入された例題をについて実際に 55 人の学生が参加し評価実験を行ってい

る。学生群を３つに分け、1）経験有で従来のチェックリストによるインスペクション、2）経験有で提案手法のアニメーション/インスペクション、3）初学生が本提案システムを使った場合に分け、バグの発見率で比較し提案手法の有効性を示している。

２．審査結果の要旨

本論文の目的は要求分析・定義過程における困難な作業を軽減させ、後段のソフトウエア設計工程にバグのない仕様を与えるためにインフォーマル仕様記述手法とフォーマル仕様記述手法を融合させた研究である。インフォーマル仕様記述とフォーマル仕様記述の２つを作成することはかなりな負担を利用者に要求することになるが、提案手法によりインフォーマル/フォーマル記述の間の追跡性を可能にし、システムが絶えず仕様間の関係を追跡することができるので、仕様の漏れや誤りを指摘し対話式に修正することが可能となる。また、フォーマル仕様記述はCDFDの形で表現されており、形式的に実行可能であるので、全ての動的な振舞い（シナリオ）をアニメーションと言う形で効率良く切り出すことができるので仕様の漏れや誤りも容易に発見可能となる。このアニメーション機能とトレースに基づくインスペクションの自動化によりフォーマル仕様記述の有用性をアピールすることができたと言える。

さらに、Ｃ＃で数万行に及ぶプロトタイプシステムを開発し、実際にバグが混入された仕様記述を用いて、多数の学生を実験に参加させバグ発見率の指標で提案手法の有効性を実証したことは、実用性の面でフォーマル手法の有用性を示したこととして意義深いと言える。また本人がプログラミング能力など幅広い技術と深い知識を有していることも窺える。

先に行われた予備審査会、および予備審査小委員会会議で指摘された事項に関しては、提出された本論文および、6月12日に行われた公聴会においていずれも、正しく加筆修正が行われていることを確認した。

以上のことより，本審査小委員会は全会一致をもって提出論文が博士（理学）の学位に値するという結論に達した。

（報告様式Ⅲ）

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

Mo Li
Graduate School of Computer and Information Sciences
Hosei University
Email: mo.li.3e@stu.hosei.ac.jp

***Abstract*—Writing formal specifications has been suggested to be effective in helping developers understand user's requirements and in 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. In order to effectively detect the defects in the formal specification, in this paper, we propose a novel and practical inspection approach for specification verification and validation. In this approach, an animation method is adopted as a reading technique to guide the reviewer to read the specification. It dynamically presents the specification to the inspector by means of "executing" it in a step-by-step manner. In each step of the animation, the inspector is required to check a group of formal specification items related to the "execution" based on a traceability-based checklist.The traceability is presented by relations between the formal specification items and their original requirements described in the informal specification, which is used to document the user's requirements using a structured natural language. In the checklist, the relations are used to raise specific questions to examine each formal specification item to check the consistency between the informal and formal specifications. A prototype supporting tool it built to support specifications construction and the inspection process. An experiment is conducted to evaluate the performance of our inspection approach, and the experiment results indicate that our inspection approach can help inspector find more bugs than the traditional checklist-based animation, especially the bugs that affect the consistency between the informal and formal specifications due to the usage of traceability-based checklist.**

I. Introduction

The role of requirement analysis in assuring the quality of software products has been well recognized and writing formal specifications has been suggested to be effective in helping developers understand user's requirements and in enhancing the quality of the requirements documentation. By using mathematically-based formal notations, formal specifications can precisely define the user's requirements. However, like the other activities in software development, the construction of formal specifications is usually error-prone in practice. Therefore, detecting errors contained in the formal specifications before it is delivered for implementation is very important.

Many techniques have been studied to verify and validate formal specification. For example, formal proof [1], model checking [2], and conventional inspection [3]. But these techniques face different challenges. The formal proof requires its user must be sophisticated and have strong mathematically background. Since the proof process is usually complex and time consuming, formal proof is too difficult to be used in practice. The well-known weakness of model checking is the state explosion problem, which is caused by the infinite state space of the target system [4]. The conventional inspection lacks detailed reading techniques to guide inspector reading through the specification.

In this paper, we propose a novel and practical inspection approach for potentially effectively verifying and validating formal specifications. The essential idea of this approach is first introduced in [5]. The approach consists a specification animation-based reading technique and a traceability-based checklist. The animation method 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. In the "execution" of a system scenario, each processes involved is "executed" in a step-by-step manner. The inspector read through the formal specification by following the animation process, in each step, he is required to checks a specific process and related formal specification items 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 formal specifications. 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 each formal specification item for checking the consistency between informal and formal specifications.

In order to evaluate the performance of our inspection approach, an experiment is conducted to compare our inspection approach with the traditional checklist-based inspection method. The experiment results indicate that our inspection approach can help inspector find more bugs than the traditional checklist-based animation, especially the bugs that affect the consistency between the informal and formal specifications due to the usage of traceability-based checklist.

The inspection approach presented in this paper has a general applicability to any of the model-based (or state-based) formal notations, such as SOFL (Structured Object-oriented

Formal Language) [6], VDM, or Z, but we choose SOFL as the formal notation for discussions, partly because of our expertise in SOFL and partly because of the fact that SOFL provides a practical three-step modelling approach by which an informal requirements specification can be gradually evolved into a formal specification and allows trace links to be built between the specifications.

The rest of the paper is organized as follows. The structure of SOFL infomal and formal specifications is introduced in Section II. The system scenario-based animation method is explained in Section III. Section IV illustrates the traceability between specifications. Section V discusses the procedure of specification inspection and a case study is given. Section VI presents a tool for supporting our inspection approach, and an experiment is reported in Section VII for evaluation purpose. Section VIII concludes the paper.

II. Structure of Specifications

Since the structure of SOFL informal and formal specifications is the foundation for performing animation and building trace links, we briefly introduce the structure of the specifications in this section.

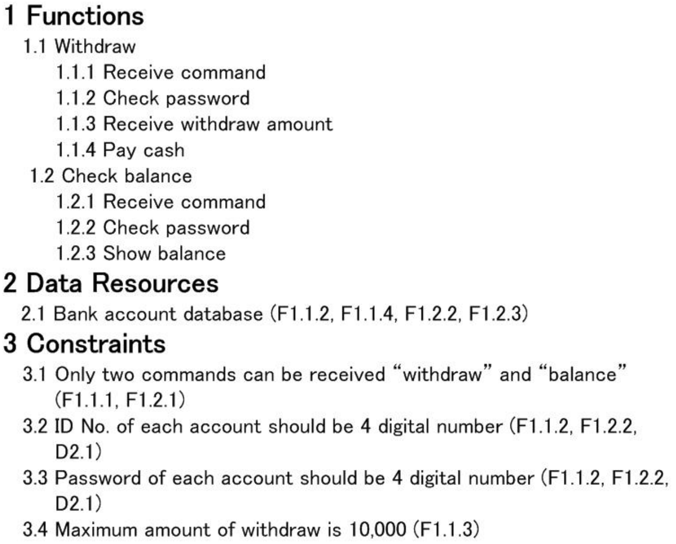
A. Informal Specification

The major task of building informal specification is to discover and collect all desired requirements from user. In SOFL modelling approach, a informal specification is written in a structured natural language. As shown in Fig. 1, the informal specification briefly describes three aspects of requirements, which include *functions*, *data resources*, and *constraints*. The functions describe the functionality that should be provided by the system; the data resources supply necessary data to the functions; and the constraints restrict the functions or data resources from different aspects. The relations between requirement items are indicated at the end of the data resource or constraint descriptions. For example, the only data resource *D2.1* will be access or update by function *F1.1.2*, *F1.1.4*, *F1.2.2*, and *F1.2.3* as shown in Fig. 1.

B. Formal Specification

A SOFL formal specification is constructed by formalizing all the requirements described in the informal specification. Related requirements are grouped into a structure called *module*. Each module may contain constant declarations, type declarations, state variable declarations, invariant definitions, and a collection of process specifications, as shown in Fig. 2. Technically, each desired function in the informal specification should be realized by a process specification; each data resource should be represented by a state variable; and each constraint is mapped to either part of a process specification or to an invariant, which is a property that must be sustained throughout the entire specification.

For each module, there is an associated formal graphical notation called Condition Data Flow Diagram (CDFD). The CDFD uses visual notation to express the relation between different processes included in the module specification. A process in a CDFD is treated as a transition and a data flow as a token. When all the input data flows of a process become available, the process will be enabled and executed. The relation between a process and a data store in CDFD is consistent with the relation between the process and the corresponding state variable in the module. The CDFD is designed especially for presenting the the architecture of the module. Fig. 3 shows the associated CDFD of the module in Fig. 2.



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 “balance” (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)

Fig. 1. A sample of SOFL informal specification

III. Animation

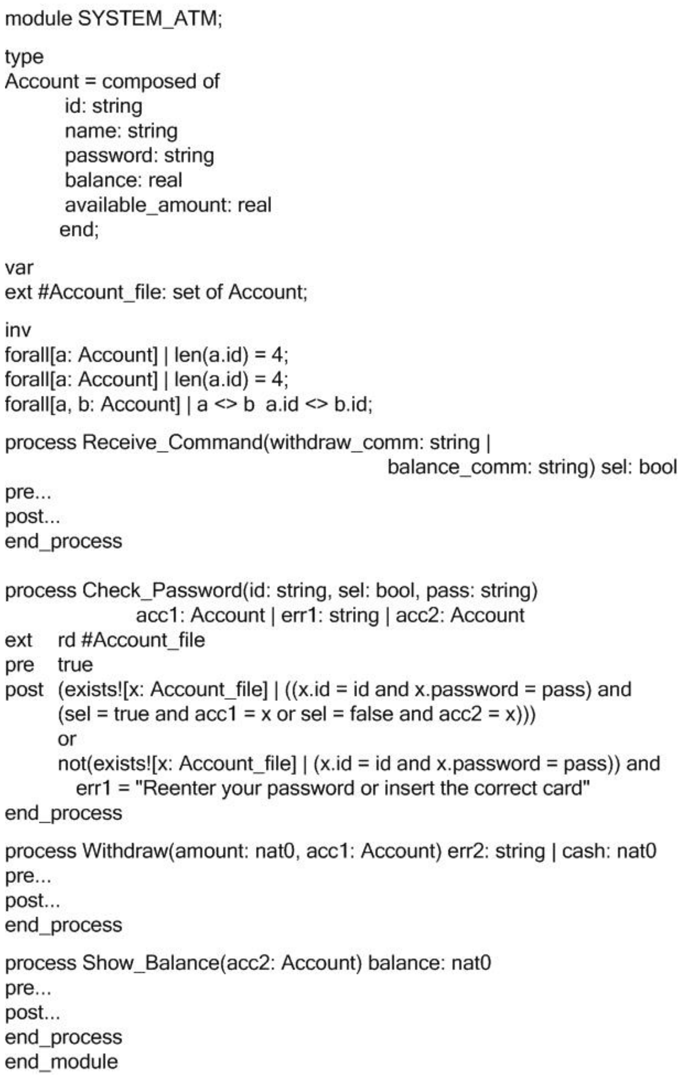
The animation method adopted in our inspection approach is called *system functional scenario-based animation method* (SFSBAM) [7]. The animation method dynamically presents the operational behaviors of the formal specification by means of “executing” corresponding system scenarios. A system scenario represents an independent operational behavior and can be formally defined as a sequence of processes or graphically presented as a data flow path in the CDFD. The formal definition of a system scenario is as follows:

Definition 1: A *system functional scenario*, or *system scenario* for short, is a sequence of processes $d_i[P_1, P_2, ..., P_n]d_o$, where d_i and d_o are the sets of input and output variables respectively, and each P_i is a process.

Based on the above definition, the CDFD shown in Fig. 3 contains the following five system scenarios:

- {*withdraw_com*}*[Receive_Command, Check_Password, Withdraw]*{*cash*}
- {*withdraw_com*}*[Receive_Command, Check_Password, Withdraw]*{*err2*}
- {*withdraw_com*}*[Receive_Command, Check_Password]*{*err1*}
- {*balance_com*}*[Receive_Command, Check_Password]*{*err1*}
- {*balance_com*}*[Receive_Command, Check_Password, Show_Balance]*{*balance*}

An algorithm has been proposed to automatically generate system scenarios based on the topology of the CDFD to facilitate the inspector. The algorithm is first implement in



```
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.id) = 4;
forall[a, b: Account] | a <> b  a.id <> b.id;

process Receive_Command(withdraw_comm: string |
                                                              balance_comm: string) sel: bool
pre...
post...
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) err2: string | cash: nat0
pre...
post...
end_process

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

Fig. 2. A sample of SOFL informal specification

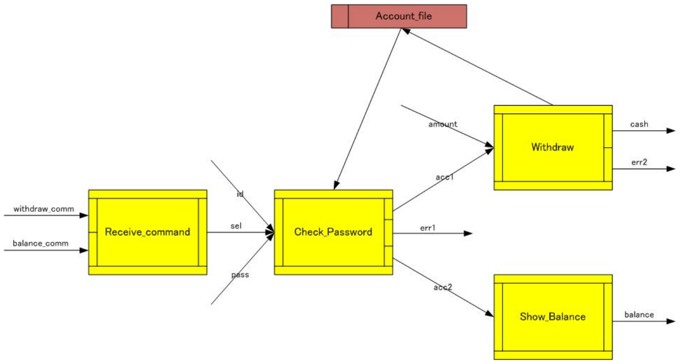


Fig. 3. The CDFD associated with the module "SYSTEM_ATM"

[8] and has been integrated to the supporting tool introduced in Section VI.

In the "execution" of a system scenario, each processes involved is "executed" in a step-by-step manner. 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 paper. 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. In the animation, the collected test suite is used to replace the corresponding input and output variables in the pre- and post-conditions to automatically check whether the pre- and post-conditions of the process can be satisfied. Only the test suite that satisfies the pre- and post conditions is used for dynamic demonstration.

Note that when a process is "executed" in the animation of a specific system scenario, only one of the *operation functional scenarios* defined in the process is involved in the "execution".

An *operation functional scenario* of a process defines an independent relation between its input and output under a certain condition. Let $P(P_{iv},\ P_{ov})[P_{pre},\ P_{post}]$ denote the process specification of a process P, where P_{iv} and P_{ov} are the sets of all input and output variables, and P_{pre} and P_{post} are the pre and post-condition, respectively. Then, the operation functional scenario is defined as follows.

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

We use $\sim x$ and x to represent the value of state variable x before and after the process respectively, and $\sim P_{pre} = P_{pre}[\sim x/x]$ denotes the predicate resulting from substituting the initial state $\sim x$ for the final state x in pre-condition P_{pre}.

For example, the process "Check_Password" defined in the formal specification shown in Fig. 2 contains the following three operation scenarios:

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"*

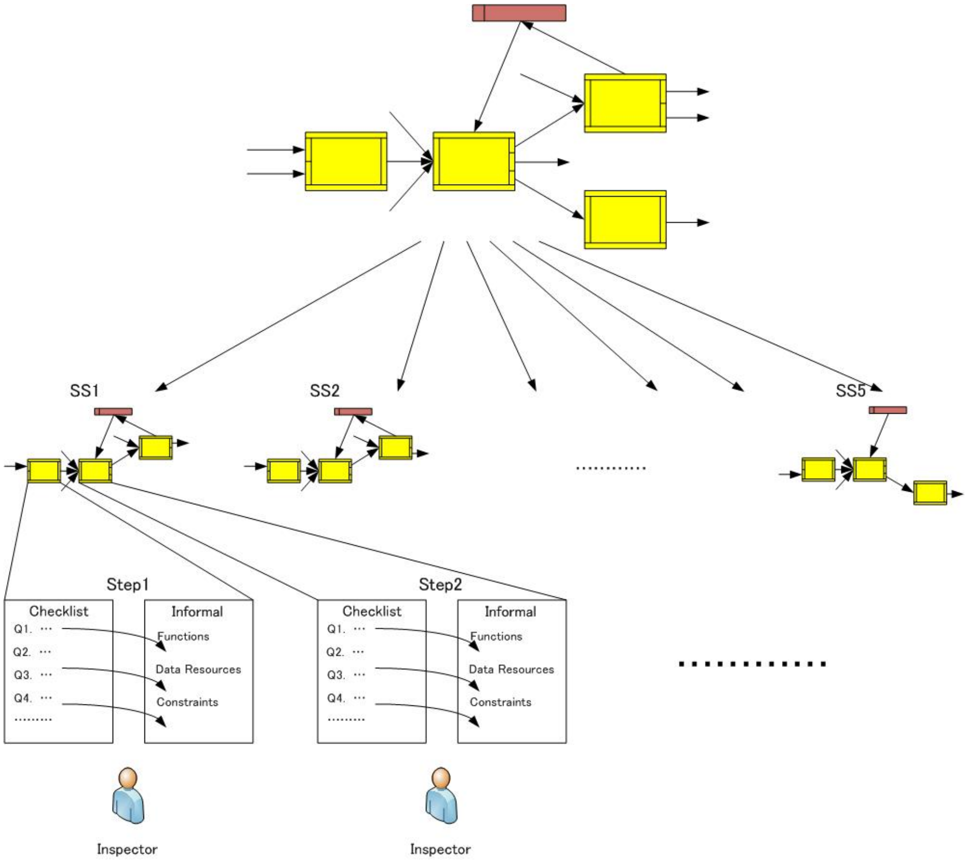


Fig. 4. The abstract procedure of formal specification inspection

If the system scenario {withdraw _ comm} [Receive _ Command, Check _ Password, Withdraw] {cash} is "executed", the operation scenario involved in the "execution" of process "Check_Password" is the first one of the three operation scenarios listed above. The test suite used to dynamically present the "execution" of process "Check_Password" must satisfy this operation scenario.

As mentioned previously, the system scenario-based animation method is adopt as a reading technique in our inspection approach. The inspector read through the formal specification by following the animation process, in each step, he is required to checks a specific process, namely a specific operation scenario, and related formal specification items against the informal specification based on a traceability-based checklist.

Fig. 4 illustrate the inspection process of using system scenario-based animation as a reading technique. The CDFD with the same structure of the CDFD shown in Fig. 3 is first divided in to five system scenarios. By following the "execution" procedure in the animation, the processes involved are usually inspected in turn. The context of the figure indicates that the first system scenario "*SS1*" is under inspected. Three processes are involved in this system scenario. By following the animation process, the first process in the system scenario is inspected in "*Step 1*". Inspector checks the formal specification items involved the current process by answering the questions on a related checklist. After the inspection of the first process, the second process in system scenario "*SS1*" is inspected in "*Step 2*". The system scenario "*SS2*" will be inspected after the inspection "*SS1*" is finished and then other system scenarios.

The checklist used in our inspection approach is traceability-based checklist, which will be introduced in the following sections.

IV. Traceability between Informal and Formal Specification

The traceability between informal and formal 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 defined the relation, the categories of the requirement items and formal specification items should be formally defined first.

A. Requirement Item

The requirement items described in the informal specification can be divided into two categories. The first category is called "*explicit requirements*" which contains the requirements defined explicitly in the informal specification. The three types of explicit requirements include functions, data resources, and constraints.

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 are defined as follows, where "*RQ*" stands for "requirement".

$RQ_1 = F_I$
$RQ_2 = D_I$
$RQ_3 = C_I$

The other category is the "*implicit requirements*" which described by the relations between explicit requirements. For instance, consider the two items: function *F1.1.2* and data resource *D2.1* in the informal specification shown in Figure 1. 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 should be realized properly in the formal specification. Three types of implicit requirements are included in the informal specification, and they are formally defined as follows:

Definition 4: Let the following three functions present the relations between different explicit requirements

$using_D : D_I \rightarrow power(F_I)$
$applyingto_D : C_I \rightarrow power(D_I)$
$applyingto_F : C_I \rightarrow power(F_I)$

Then, the three types of the implicit requirements are defined as follow:

$RQ_4 = \{(d, f) \mid d \in D_I \wedge f \in F_I \wedge f \in using_D(d)\}$
$RQ_5 = \{(c, d) \mid c \in C_I \wedge d \in D_I \wedge d \in applyingto_D(c)\}$
$RQ_6 = \{(c, f) \mid c \in C_I \wedge f \in F_I \wedge f \in applyingto_F(c)\}$

B. Inspection Target

As introduced previously, in each step of the animation, the inspector is required to check a specific operation scenario and related formal specification items. In our inspection approach, each formal specification item that needs to be inspected is called an *inspection target*. The inspection targets of a formal specification are divided into two classes. The first class is the formal specification items defined explicitly, including operation scenario, state variable, type declaration, and invariant. The second class is the dependence relations between the formal specification items. Since different explicitly defined items are working together to present the user's requirements, the dependence relations should also be inspected for their validity.

Definition 5: 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 6: 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, 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 specification inspection, and both the process and system scenario can be transformed to a conjunction or disjunction of operation scenarios.

Definition 7: Let $OS_F = \bigcup\limits_{i=1}^{n} M_i.OS_M$ be the set of all operation scenarios defined in the entire formal specification, $T_F = \bigcup\limits_{i=1}^{n} M_i.T_M$, $V_F = \bigcup\limits_{i=1}^{n} M_i.V_M$ be the set of all type and state variable declarations respectively, and $I_F = \bigcup\limits_{i=1}^{n} M_i.I_M$ 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 "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 dependency relation. Inspecting dependency relations aims to check whether formal specification items are used appropriately according to the requirements. The inspection targets that are presented by dependency relations in the formal specification are formally defined as follows

Definition 8: Let the following three functions present the dependency relations.

$using_V : V_F \rightarrow power(OS_F)$
$typeof_V : V_F \rightarrow T_F$
$typeof_{OS} : OS_F \rightarrow power(T_F)$
$applyingto_V : I_F \rightarrow power(V_F)$
$applyingto_{OS} : I_F \rightarrow power(OS_F)$

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

$IT_5 = \{(v, os) \mid v \in V_F \wedge os \in OS_F \wedge os \in using_V(v)\}$
$IT_6 = \{(v, t) \mid v \in V_F \wedge t \in T_F \wedge t = typeof_V(v)\}$
$IT_7 = \{(os, t) \mid os \in OS_F \wedge t \in T_F \wedge t \in typeof_{OS}(os)\}$
$IT_8 = \{(i, v) \mid i \in I_F \wedge v \in V_F \wedge v \in applyingto_V(i)\}$
$IT_9 = \{(i, os) \mid i \in I_F \wedge os \in OS_F \wedge os \in applyingto_{OS}(i)\}$

C. Construction of Traceability

A trace link is the connection between a inspection targets in the formal specification and its original requirement in the informal specification. The trace links in Table. 1 formally define the traceability between informal and formal specifications, which is first proposed in [9].

The traceability rule $link_1$ in the first row of Table. 1 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.

Another type of explicit requirements is data resource, which is usually realized by a state variable in the formal specification. In our inspection approach, a pair of state variable and its type will be inspected against the data resource. The rule $link_2$ in Table. 1 represents the trace link from a pair of state variable and its type to a data resource.

The third type of explicit requirement is constraint. In practice, a constraint is realized either by an invariant or by

TABLE 1

TRACEABILITY RULES BETWEEN INSPECTION TARGETS AND REQUIREMENT ITEMS

ID	Review Target(Description)	Requirement(Description)	Definition
$link_1$	$IT_1(operation\ scenario)$	$RQ_1(function)$	$link_1 : IT_1 \rightarrow power(RQ_1)$
$link_2$	$IT_6((state\ variable, type))$	$RQ_2(data\ resource)$	$link_2 : IT_6 \rightarrow RQ_2$
$link_3$	$IT_4(invariant)$	$RQ_3(constraint)$	$link_3 : IT_4 \rightarrow RQ_3$
$link_4$	$IT_1(operation\ scenario)$	$RQ_3(constraint)$	$link_4 : IT_1 \rightarrow RQ_3$
$link_5$	$IT_5((state\ variable, operation\ scenario))$	$RQ_4((data\ resource, function))$	$link_5 : IT_5 \rightarrow RQ_4$
$link_6$	$IT_8((invariant, state\ variable))$	$RQ_5((constraint, data\ resource))$	$link_6 : IT_8 \rightarrow RQ_5$
$link_7$	$IT_9((invariant, operation\ scenario))$	$RQ_6((constraint, function))$	$link_7 : IT_9 \rightarrow RQ_6$

part of a process. The invariants usually realize the constraints used to restrict data resources, and the constraints for functions are generally realized as a part of process. The rules $link_3$ and $link_4$ in Table. 1 define the trace links from an invariant or operation scenario to the corresponding constraint.

The three types of implicit requirements are described as the relations between different explicit requirements. Similarly, the implicit requirements are realized in the formal specification as dependence relations. The traceability rules $link_4$, $link_5$, and $link_6$ in Table. 1 describe the trace links from different dependence relations to corresponding implicit requirements.

V. FORMAL SPECIFICATION INSPECTION

To inspect a formal specification, the inspector should examine each inspection from four aspects: necessity, appropriateness, correctness, and completeness.

A. Necessity

Checking the necessity property aims to ensure that no declared type identifier, state variable, or invariant is not used in the process specifications of each module.

B. Appropriateness

Appropriateness requires that the inspection targets realize the original requirements in an appropriate manner. Since the appropriateness of some kinds of inspection targets cannot be formally defined, examining whether they are appropriate highly depends on human judgement. For example, the state variable "*Account_file*" is defined in the formal specification in Fig. 2 with a type "*set of Account*". This state variable and its type realize the data resource *D2.1* described in the informal specification in Fig. 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 represents the essence of the data resource?" and "Whether the type of the state variable is appropriate for the data resource?".

Below are some appropriateness-related properties that can be formally defined.

Property 1: If the dependency relation of RT_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 review 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 dependency relations of review targets RT_9 and RT_{10} should satisfy the following two properties.

Property 2: If the dependency relation of RT_9 is appropriate, the following predicate must hold:

$$\forall_{i\in I_F, v\in V_F, c\in C_I, d\in D_I} \cdot (c,d) = link_6((i,v)) \Rightarrow$$
$$c \in link_3(i) \wedge \exists_{t\in T_F} \cdot (t = typeof_V(v) \wedge d = link_2((v,t)))\wedge$$
$$v \in applyingto_V(i) \wedge d \in applyingto_D(c)$$

Property 3: If the dependency relation of RT_{10} is appropriate, the following condition must hold:

$$\forall_{i\in I_F, os\in OS_F, c\in C_I, d\in D_I} \cdot (c,f) = link_5((i,os)) \Rightarrow c \in link_3(i)$$
$$\wedge f \in link_1(os) \wedge os \in applyingto_{OS}(i) \wedge f \in applyingto_F(c)$$

C. 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 Property 4 and 5 below.

Property 4: 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 previous property, 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.

Property 5: Let one of related invariant on type T be $I_t = \forall_{t\in T} \cdot Q(t, w)$. Then the following predicates must hold:

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

2)$\sim P_{pre} \wedge C_i \wedge D_i \Rightarrow \forall_{v\in T} \cdot Q(t, w)[v/t]$

Since the invariants and the operation scenarios are defined separately, performing syntax checking cannot ensure the variables in the operation scenario comply with corresponding invariants. The property states that when the pre- and post-conditions of an operation scenario evaluates to true, the related invariants must hold before and after the execution of the operation scenario.

D. Completeness

Checking completeness reveals whether all the user's requirements are realized completely. According to the

TABLE 2
INSPECTION OF A SPECIFIC SYSTEM SCENARIO

NO.	Question	Answer
1	Does operation scenario "*true and exists![x: Account_file] | (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	Is this operation scenario specified complying with SOFL?	Yes
5	Does this operation scenario appropriately realize function $F1.1.2$?	Yes
6	Whether variable "sel", "id", "$pass$" and "$Account\_file$" realize any data resource?	Yes
7	Which variable realize which data resource?	"*Account_file*" realizes $D2.1$
8	Does the variable "*Account_file*" and its type "*set of Account*" appropriately realize the data resource $D2.1$?	Yes
9	Does the function $F1.1.2$ use data resource?	Yes, $D2.1$
10	Does this operation scenario use the state variable that realizes the data resource?	Yes, "$Account\_file$"

definition of the traceability rule between function items and operation scenarios, each function item in the informal specification may be realized by more than one operation scenario in the formal specification. The inspector can hardly make any judgement of completeness by merely inspecting each operation scenario independently. The operation scenarios realizing the same function item should be well organized and inspected as a whole to ensure the function is formalized completely. 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.

E. Case Study

We presents a case study to demonstrate how our inspection method works in practice. Fig. 2 and 3 show the selected formal specification in our case study. The corresponding requirements are documented in the informal specification as shown in Figure 1.

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. Since all of the types, state variables, and invariants declared are used somewhere in the formal specification, property "necessity" is satisfied.

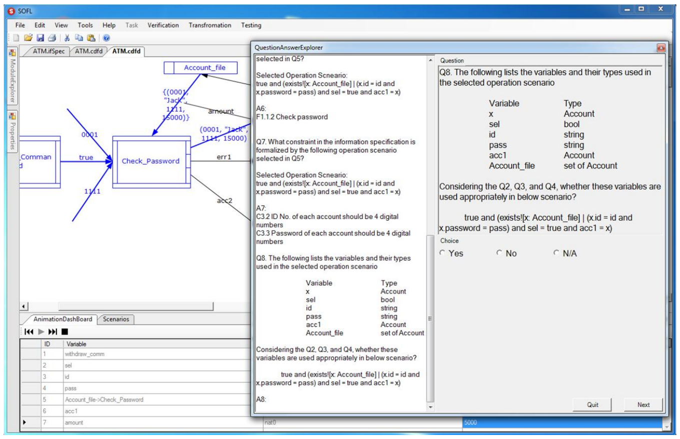
The second stage is to inspect the formal specification by following the animation process. At this stage, the inspector should answer the questions raised based on the traceability, appropriateness, and correctness. Suppose the system scenario $\{$withdraw _ comm$\}$ [Receive _ Command, Check _ Password, Withdraw] $\{$cash$\}$ is selected for inspection, Table. 2 lists up the questions for inspecting the operation scenario of process "Check _ Password" and related inspection targets. For the sake of space, only some typical questions are listed in Table 2; the inspection of others can be understood in the same way.

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 "$F1.1$".

VI. TOOL SUPPORT FOR SPECIFICATION CONSTRUCTION AND INSPECTION

A prototype software tool we have developed to support the entire procedure of our formal specification inspection approach. Actually, the tool is developed not only for supporting the inspection method, but for providing a framework to support the entire SOFL three-step modelling approach and related techniques. It is implemented in C# programming language under the environment of Microsoft Visual Studio 2008. The major functions provided by the current version of the framework is summarized as follows:

1) **Specifications Organization** All kinds of SOFL specifications constructed and managed in our framework. Each specification is saved in an independent XML file and the hierarchy of the specification is save in a file with suffix ".soflpro", which stands for SOFL project.
2) **Formal Specification Editor** A customized editor is provided for SOFL specification construction. This editor consists of a draw board for drawing CDFD, an



ID	Variable
1	withdraw_comm
2	sel
3	id
4	pass
5	Account_file->Check_Password
6	acc1
7	amount

nat0 5000

selected in Q5?

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

A6:
F1.1.2 Check password

Q7. What constraint in the information specification is formalized by the following operation scenario selected in Q5?

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

A7:
C3.2 ID No. of each account should be 4 digital numbers
C3.3 Password of each account should be 4 digital numbers

Q8. The following lists the variables and their types used in the selected operation scenario

Variable	Type
x	Account
sel	bool
id	string
pass	string
acc1	Account
Account_file	set of Account

Considering the Q2, Q3, and Q4, whether these variables are used appropriately in below scenario?

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

A8:

Q8. The following lists the variables and their types used in the selected operation scenario

Variable	Type
x	Account
sel	bool
id	string
pass	string
acc1	Account
Account_file	set of Account

Considering the Q2, Q3, and Q4, whether these variables are used appropriately in below scenario?

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

○ Yes ○ No ○ N/A

Fig. 5. The snapshot for specification inspection

editor for writing formal specification, and a function to automatically keep consistency between CDFD and formal specification.

3) **Specification Animation** The system scenarios will be automatically extracted based on the topology of the CDFD, and the animation performing on the CDFD can be manipulated in a step-by-step manner
4) **Inspection** The questions for each inspection targets can be automatically constructed based on a built-in question template. Each question will be ask in an interactive manner. Fig. shows the GUI for specification inspection.

Moreover, our framework is designed as a platform to easily integrate other functions. For now, several functions have been integrated into this framework.

VII. Experiment

An experiment has been conducted to evaluate the performance of our inspection approach. In the experiment, the subjects are divided into three groups based on their experience in SOFL. Each group is required to inspect the same formal specification by using either our inspection approach or the traditional checklist-based inspection method. By carefully analyzing the experiment results, we make several conclusions. For the sake of space, we list tow major conclusions as follows:

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, especially the defects that affect the consistency between informal and formal specifications.

Conclusion 2: The ability of an inspector can affect the results of an inspection. The inspector with more experience and better skills will make better results, and using a more effective inspection method can help to offset the difference caused by the lack of experience.

By interviewing the subjects after the experiment, we find that our inspection approach does provide an effective reading technique for inspection, however, the duplication of questions in the checklist may affect the efficiency of our approach.

VIII. Conclusion

In this paper, we propose an novel inspection approach to verify and validate the formal specification. A system scenario-based animation method proposed to help the inspector read the formal specification. The traceability between requirement items and inspection targets are formally defined. Traceability-based checklist helps the inspection examine the consistency between informal and formal specifications. In our inspection approach, each inspection target is required to be reviewed from four aspects: necessity, appropriateness, correctness, and completeness. An case study is presented to illustrate the inspection process. A supporting tool has been developed to support SOFL specifications construction and specification inspection. An experiment is conducted to evaluate the effectiveness of our approach by comparing it to the traditional checklist-based inspection approach. In the future, we plan to optimize the procedure of our inspection approach to make it more efficient, and we also want to build a checklist base to allow inspectors create their own checklists and reuse the existing checklists.

Acknowledgement

I want give my sincere gratitude to my supervisor Prof. Shaoying Liu for his great support throughout my research work. His enthusiasm for researching inspired me, and his encouragement helps me to overcome many difficulties.

References

[1] O. Grumberg E. M. Clarke and D. Peled. Model checking. MIT press, 1999.

[2] M. E. Fagan. Design and code inspections to reduce errors in program development. IBM Systems Journal, 15(3):182.211, 1976.

[3] N. Shankar S. Owre, J. M. Rushby and F. Von Henke. Formal verification of fault-tolerant architectures: Prolegomena to the design of pvs. IEEE Transactions on Software Engineering, 21(2):107125, 1995.

[4] K. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic Publisher, 1993.

[5] Mo Li and Shaoying Liu, "Traceability-Based Formal Specification Inspection," Proceedings of Eighth International Conference on Software Security and Reliability (SERE), IEEE Press, San Francisco, USA, 30 June-2 July, 2014, pp.167-176.

[6] S. Liu. Formal Engineering for Industrial Software Development Using the SOFL Method. Springer-Verlag, ISBN 3-540-20602-7, 2004.

[7] Mo Li and Shaoying Liu, "Automated Functional Scenarios-Based Formal Specification Animation," Proceedings of 19th Asia-Pacific Software Engineering Conference (APSEC), IEEE Press, Hong Kong, 4-7 Dec. 2012, pp.107-115.

[8] Mo Li and Shaoying Liu, "Tool Support for Rigorous Formal Specification Inspection," Proceedings of IEEE 17th International Conference on Computational Science and Engineering (CSE), IEEE Press, Chengdu, China, 19-21 Dec., 2014, pp.729-734.

[9] Mo Li and Shaoying Liu, "Reviewing Formal Specification for Validation Using Animation and Trace Links," Proceedings of 21th Asia-Pacific Software Engineering Conference (APSEC), IEEE Press, Jeju, Korea, 1-4 Dec., 2014, pp.286-293.