# Integrating Animation-Based Inspection Into Formal Design Specification Construction for Reliable Software Systems

Mo Li and Shaoying Liu, *Senior Member, IEEE*

***Abstract*—Software design has been well recognized as an important means to achieve high reliability, and formal specification can help enhance the quality of design. However, communications between the designer and the user can become difficult via formal specifications due to the potentially complex mathematical expressions in the specification. This difficulty may lead to the situation where the user may not be closely involved in the process of constructing the specification for quality assurance. To allow formal specification to play more effective roles in software design, we put forward a new approach to deal with this problem in this paper. The approach is characterized by integrating specification animation-based inspection into the process of constructing formal design specifications. We discuss the underlying principle of the approach by explaining how specification animation is utilized as a reading technique for inspection to validate, and then evolve, the current specification towards a satisfactory one. We describe a prototype software tool for the method, and present a case study to show how the method supported by the tool works in practice.**

***Index Terms*—Formal specification, specification animation, specification evolution, verification and validation.**

## NOTATION

| | |
|---|---|
| $RQ$ | a set of requirements. |
| $IT$ | a set of inspection targets. |
| $P(x; y; s)$ : $[pre\_P, post\_P]$ | a process specification, where $x$, $y$, and $s$ are the sets of input data flow variable declarations, output data flow variable declarations, and external variable declarations, respectively; $pre\_P$, and $post\_P$ denote the pre-, and post-conditions of process $P$, respectively. |
| $len(list)$ | the length of the sequence $list$. |

## ACRONYMS AND ABBREVIATIONS

| | |
|---|---|
| **CDFD** | Condition Data Flow Diagram |
| **VDM-SL** | Vienna Development Method—Specification Language |
| **SOFL** | Structured Object-oriented Formal Language |
| **SFSBAM** | System functional scenario-based animation method |
| **ATM** | Automated Teller Machine |
| **Tcl/Tk** | Tool Command Language/Tool kit |

Manuscript received September 12, 2014; revised December 17, 2014; accepted July 10, 2015. . This work was supported in part by JSPS KAKENHI Grant Number 26240008 and in part by the SCAT research foundation. Associate Editor: W. E. Wong.

M. Li is with the Graduate School of Information Sciences, Hosei University, Koganei, Japan.

S. Liu is with the Department of Computer Science, Faculty of Computer and Information Sciences, Hosei University, Koganei, Japan.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2015.2456853

## I. INTRODUCTION

SOFTWARE design has been widely recognized as an important technique for the assurance of the reliability of software systems in the software engineering community [1], [2]. The effectiveness of design is heavily determined by specific design methods, and notations for writing design documents, known as *design specifications*. While most of the existing design methods, such as data flow design [3], structured design [4], and object-oriented design [5]–[7], use natural language, graphical notation, or the combination of both for writing design specifications, formal methods advocate the use of mathematically-based notation for specification [4], [8]–[10]. Although the effectiveness of formal methods in realistic systems development is controversial [11]–[13], the latest survey conducted by Woodcock and his colleagues in [14] indicates that some industrial groups working in the domain of safety critical systems find formal specification useful in helping them obtain sufficient understanding of the envisaged system. Meanwhile, the survey also points out that formal verification techniques are rarely used in realistic projects.

While recognizing the useful effect of formalization, we find that formal notation alone is unlikely to be used in industry because most of the practitioners do not have a strong mathematical background, and development activities are constrained by limited budget and time [15]. A more practical formal notation, known as Structured Object-oriented Formal Language (SOFL), has therefore been developed for improvement [16], [17].

SOFL is an integrated specification language that uses a formalized data flow diagram notation called *condition data flow diagram* (CDFD) to describe the architecture of the system, and a text-based mechanism called *module* to formally define the components of the CDFD, including *data flows*, *data stores*, and *processes* (or operations in general terms), using a formal notation similar to the Vienna Development Method—Specification Language (VDM-SL). Such an architecture-based approach to constructing formal specifications has been found to be suitable for the abstract design of software systems, and helpful in reducing changes in formal specifications [16], [18]–[20]. Although this approach has enhanced the comprehensibility of design specifications for communication due to the use of the visualized notation CDFD, communication via the entire formal

specification still faces difficulty at the detailed level because of the mathematical expressions used in the module. As indicated in the principles of *agile* development methods [21], an important factor to ensure the quality of the developed software systems is the active involvement of the user throughout the development process. We believe that this observation is also applicable to formal specification construction according to our experiences in many software projects using SOFL [19], [22], but the problem is how the user can be actively brought into the process of formal specification construction.

In this paper, we put forward a new approach, aiming to tackle this problem. The approach features the incorporation of specification animation-based inspection into the process of constructing formal specifications. By specification animation-based inspection we mean an inspection method that uses specification animation as a reading technique for inspection. Reading is generally the essential technique adopted in inspection, and various reading techniques have been well studied [23]–[30]. 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 been found to be beneficial, as discussed in detail from Section IIto Section V.

This method can be incorporated into the process of specification construction to enhance its quality. As briefly mentioned previously, constructing a SOFL specification usually starts with the construction of the architecture using CDFD, and then proceed to the definition of all the components of the CDFD in the corresponding module. Processes in a high level CDFD can be decomposed into a low level CDFD to which a corresponding module needs to be built for the same purpose as its parent module. Repeating the decomposition of high level processes at various levels of CDFDs will lead to a hierarchy of CDFDs, and their corresponding modules. The principle underlying our proposed approach is to apply the animation-based inspection to verify and validate every level of CDFD and its associated module before any decomposition of the processes in the CDFD is undertaken. It can also be used as a communication means to obtain feedback from the user for further evolution and improvement of the specification.

In this paper, we make the following two major contributions.

- 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 functional scenarios 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 a specific relationship between the input and output of the CDFD through the formal specifications of the processes involved in the path; see details in Sections III, and IV.
- 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 valid 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 functional scenario, and managing all of the related data files.

Apart from these two major technical contributions, we also present a case study conducted to check the usability and potential effectiveness of the tool supported approach. Although the case study is not conducted in the industrial setting due to many practical constraints to serve as a systematic evaluation of our approach, it allows us to observe and experience the major important aspects of the tool support and the potential performance of the approach, which we believe will interest researchers and practitioners in software engineering. It also helps us reveal areas that need further improvement.

Note that the principles of the proposed approach in this paper are not only applicable to the SOFL specification language, but also to other similar model-based formal notations, such as the well known VDM-SL [31] and B-Method [32]. We choose SOFL as the target formal technique for discussion in this paper partly because it has been recognized as a practical technique by academics and practitioners [19], [33]–[35], 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.

The rest of the paper is organized as follows. Section II describes the underlying principle of SFSBAM through which all of the activities necessary for the method are explained abstractly. Section III discusses the activities necessary for specification animation. Section IV explains how the animation of a single system functional scenario can be performed as a reading technique to guide the inspector to carry out an inspection of system functional scenarios. Section V describes the prototype tool in terms of its architecture, functionality, features, and snapshots of applications. Section VI presents a case study that shows how the proposed approach works in practice. Section VII compares our work to existing related work. Finally, in Section VIII, we conclude the paper, and point out future research directions.

## II. Principle of Integrating Animation-Based Inspection Into Specification Construction

A flexible, practical way to use the SOFL method is to construct a formal specification directly from the informal specification obtained from a systematic requirements analysis process [17]. An informal specification is usually written in a natural language and intended to document the user's requirements. It contains three sections: *functions*, *data resources*, and *constraints*. These sections are shown in the exemplified

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)

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, F1.2.1, D2.1)
- 3.3 Password of each account should be 4 digital number (F1.1.2, F1.2.1, D2.1)
- 3.4 Maximum amount of withdraw is 10,000 (F1.1.3)

Fig. 1. Example of a simplified informal specification for ATM software.

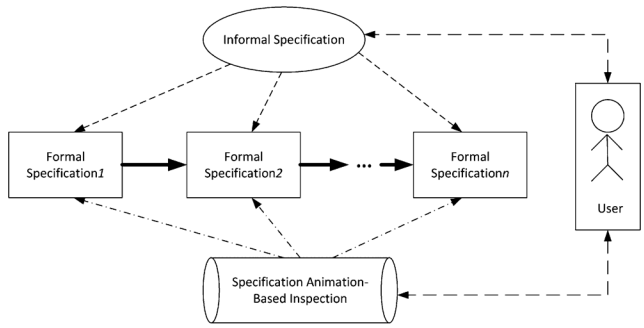


Fig. 2. Illustration of the principle of integrating the animation-based inspection into the process of formal specification construction.

informal specification for a simplified version of Automated Teller Machine (ATM) software in Fig. 1. A function describes a desired behavior to be implemented in the system under development. A data resource indicates a data item that needs to be used by a function. 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.

As mentioned briefly in the previous section, a formal specification consists of a hierarchy of CDFDs, and the associated hierarchy of modules, which is usually used for abstract design of software systems. While a CDFD, a formalized data flow diagram with a well defined operational semantic [36], [37] is used to define the architecture of the system by describing how operations (called *processes* in SOFL) are connected using data flows and data stores, and the semantic details of every component of the CDFD are formally defined using a formal notation in an associated module. This approach will make the specification comprehensible in both construction and reading [38].

The contents of the formal specification usually result from several sources, including the given informal specification, and the designer's decisions obtained through communications with the user. Basically, we assume that the informal specification represents all the user's requirements accurately and completely. But because the requirements are usually described in natural language that cannot avoid ambiguity, formalizing them during the construction of the formal specification may need further decisions for interpreting or refining some of the informal requirements. These decisions are often made by the designer alone because the user is usually hard to involve in the details of the formal specification. For this reason, there is a strong need to validate the constructed formal specification. The question is how to do the validation so that the user can be easily and actively involved in the process of constructing the formal specification to prevent errors as early as possible.

We propose to integrate specification animation-based inspection into the process of formal specification construction to tackle this problem. As illustrated in Fig. 2, the principle underlying our approach is to integrate inspection and specification construction in an interleaving manner. The very first version of the formal specification, called Specification 1, is written based on the informal specification. When one level CDFD and its associated module are constructed, specification animation will be carried out as a reading technique to guide the inspection for validating the specification against the informal specification in which the user can be involved for judgment. On the basis of the animation-based inspection, the current specification is further evolved into another version, called Specification 2, on the basis of the informal specification, the feedback from the user obtained during the animation, and possibly the designer's engineering judgment. The newly developed CDFD and its associated module in Specification 2 are animated again for validation in a similar manner. This kind of interleaving process continues until the satisfactory version, called Specification $n$, is completed. Note that the process of constructing the specifications in Fig. 2 is usually not a strict refinement as defined by Morgan [39]; it is rather an evolutionary process where the later specifications may be an extension or modification of the previously built specifications. Such a relation can hardly be formalized mathematically.

Two issues in the proposal are important.

One is how the animation should be performed to facilitate the inspection. The other is how the current specification should be further developed or evolved into another version. Before going into the details later in this paper, we explain our essential ideas here.

Specification animation is performed for a CDFD and its associated module each time to facilitate inspection. The first step is to derive all functional scenarios from the CDFD, known as *system functional scenarios*. Each scenario is supposed 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 [40]. The second step is to select an appropriate *test case* (or animation case), and the expected output for animation.

The third step is to use the selected test case and the expected output to demonstrate the scenario by demonstrating the input-output relationship of every process in turn involved in the scenario. Such an animation is used as a reading technique to guide the inspection to check the current specification.

Formal specification construction can be done using existing SOFL techniques, such as top-down, bottom-up, or middle-out approaches [17], [41], [42]. Using the top-down approach, the

top-level CDFD is first built, and its associated module is defined. This should be a formalization involving the top-level functions in the informal specification. Further development of the current specification will be done by decomposing high level processes into low level CDFDs and their associated modules, which should be kept consistent with the decomposition of the corresponding high level functions in the informal specification. By continuing such a decomposition throughout the whole level CDFDs in the current hierarchy, accompanied by the animation-based inspection, a satisfactory formal specification can be eventually constructed.

The bottom-up approach presents an opposite direction of developing CDFDs and their associated modules. The bottom level CDFDs are first built, and their modules are specified. These CDFDs are then composed to form higher level CDFDs and their modules. Continuing this process will eventually lead to the top-level CDFD and its module under which all level CDFDs and modules are organized as a system. The middle-out approach starts with middle level CDFDs and modules, and then applies both top-down and bottom-up approaches, respectively, to form the entire hierarchy of CDFDs and their associated modules. Because these techniques for constructing a formal specification in SOFL have been reported in many of our previous publications as mentioned above, the focus of this paper will be put on animation-based inspection for specification validation.

## III. Specification Animation Approach SFSBAM

In this section, we describe how a specification animation can be done in SFSBAM by focusing on the three steps of animation, as mentioned previously.

### A. Derivation of System Functional Scenarios

System functional scenarios are derived from a CDFD describing the architecture of the system at the present level. Each scenario describes a data flow path from a starting process to a terminating process. The following definition indicates how a 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, $d_i[P_1, P_2, \ldots, P_n]d_o$, where $d_i$ is the set of input variables of the scenario, $d_o$ is the set of output variables, and each $P_i$ $(i \in \{1, 2, \ldots, n\})$ is a process, but $P_1$ must be a starting process, and $P_n$ must be a terminating process. 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* $d_i[P_1, P_2, \ldots, P_n]d_o$ defines a specific behavior that transforms input data item $d_i$ into the output data item $d_o$ through a sequential execution of processes $P_1, P_2, \ldots, P_n$. 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, the CDFD of a simplified ATM is given in Fig. 3, which contains four processes. The process *Receive_Command* receives one of the two input commands, and produces one output data flow *sel* indicating which command is received. The process *Check_Password* then checks whether the input cards *id* and *pass* are correct with respect to the registered ones in the data store *Account_file*. If confirmed to be correct, the process will then produce either output data flow *acc1* or *acc2*, depending on the value of the input data flow *sel*. If *sel* indicates the command being the *withdrawal* command, then *acc1*, which contains the customer's account information, is produced and sent to the process *Withdraw*; otherwise, *acc2* is produced and sent to the process *Show_Balance*. The process *Withdraw* offers the function to withdraw the requested money from the customer's account, and the process *Show_Balance* provides the account balance to the customer. From this CDFD, the following five system scenarios can be derived.

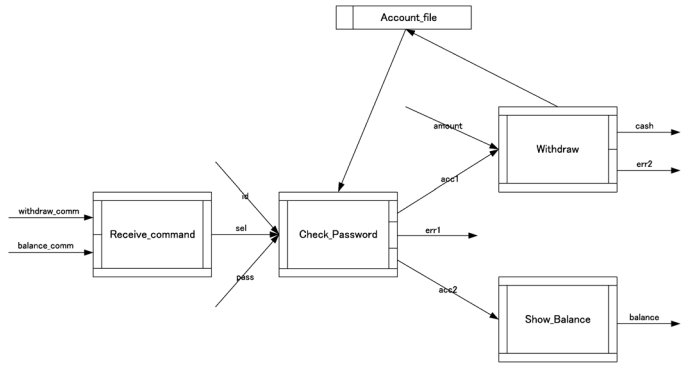


Fig. 3. CDFD of simplified ATM software.

- $\{withdraw\_comm\}[Receive\_Command,\ Check\_Password, Withdraw]\{cash\}$
- $\{withdraw\_comm\}[Receive\_Command,\ Check\_Password, Withdraw]\{err2\}$
- $\{withdraw\_comm\}[Receive\_Command,\ Check\_Password]\{err1\}$
- $\{balance\}[Receive\_Command,\ Check\_Password]\{err1\}$
- $\{balance\}[Receive\_Command,\ Check\_Password,\ Show\_Balance]\{balance\}$

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 scenarios can be interpreted similarly.

To interpret the semantics of the system scenario, we must understand the semantics of the processes and data items involved. In the SOFL formal specifications, such semantic details are defined in modules. Each module is always associated with a CDFD in the sense that the CDFD describes how the components, such as processes, data flows, and data stores, are connected to form the entire system, while the associated module provides formal definitions of the components of the CDFD. The CDFD and module has a strict one-to-one relationship. For example, an outline of the module associated with the CDFD in Fig. 3 is given in Fig. 4

In our previous work [43], we have worked out an algorithm for automatically deriving all of the system scenarios from a CDFD. The essential idea of the algorithm is first to decompose the CDFD into a graph in which all of the input ports receiving input data flows and output ports sending output data flows of a single process can be clearly related, and then to use a depth-first search algorithm to find all possible paths from the input ports of the starting processes to the output ports of the terminating

```
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: 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. 4. Example of formal specification in a module.

processes. Because the content of the algorithm has already been made available in the literature, we do not explain it further here. The reader can refer to our previous publication [43] for details.

### B. Test Suite Selection

Animation of a CDFD is performed by animating each individual system scenario derived. Animation of a scenario in this case is actually a dynamic demonstration of the behavior of the scenario in terms of showing what input leads to what output.

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*. Because 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. Both the designer and the user may also need to work together to determine whether faults are found by the inspection.

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 respectively of the process to be animated. Only the satisfied test suite is used for dynamic demonstration. Although a full automation in test suite generation can be possible [44], 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 the designer. For internal consistency checking of a system scenario, an 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.

TABLE I
TEST CASE 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 |

A test suite should be generated to fulfill at least the following targets 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, or 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 a test suite manually to demonstrate a normal use case in Table I.

### C. Execution of Functional Scenario

Demonstration of a system scenario is to dynamically display the situation of how the test case is taken as input to generate the expected result for each process of the scenario. Such a demonstration 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. Fig. 5 shows a process of animating the first system scenario derived from the ATM CDFD. Fig. 5(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 diagrams together depict the process of the animation of the system scenario. The test cases and the expected results used in this animation are taken from Table I.

Note that the animation only provides an example of executing the scenario, but is not capable of detecting any errors by itself. It is humans (e.g., the designer, the user, or anybody

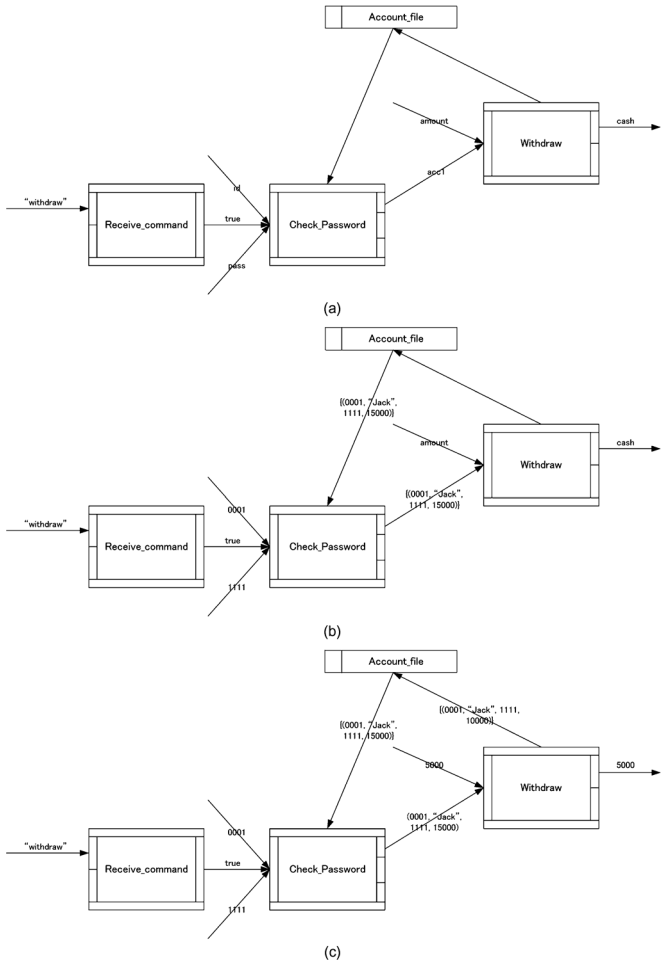


Fig. 5. The process of animating a system scenario.

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. Our experience with many software development projects over the last thirty years suggests that a practical solution is software inspection, but 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 next.

## IV. Utilizing SFSBAM for Inspection

In this section, we describe how SFSBAM can be used to assist in the inspection of a formal specification. The essential idea is to use SFSBAM to guide the human inspector to inspect all of the interesting aspects of the target system scenario based on a checklist prepared in advance. In the previous discussion, we have already shown how an animation of a system scenario can display the situation in which the input data flows are transformed into the output data flows through a step-by-step manner of demonstrating all of the involved processes sequentially.

This process can be utilized to guide the inspector to inspect the scenario in a step-by-step manner. The left unaddressed problem in our approach is how to carry out inspection based on a prepared checklist. In this section, we will focus on the design of the checklist, and its application to inspections.

The questions on the checklist will be formed based on the traceability from the formal specification to the informal specification, and the desired properties the formal specification must possess. The questions raised from the traceability are expected to help check whether all the requirements described in the informal specification are formalized in the formal specification, and the questions raised based on the properties of the formal specification cover four aspects of the specification: *necessity*, *appropriateness*, *correctness*, and *completeness*. The user can check whether the formalization of the requirements is correct and accurate by answering these questions. The details will be discussed in Sections IV-A and IV-B.

### A. Traceability and Inspection Targets

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 an appropriate checklist for inspection, we must figure out what parts in both specifications should be connected. This activity 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.

*1) Explicit and Implicit Requirements:* The connections between two specifications mentioned above can be separated into two categories. The first kind of connection is called *connection of explicit requirement*, 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 item* to mean any requirement item that is explicitly described in the informal specification, including functions, data resources, and constraints.

The other category of connection is called *connection of implicit requirement*, 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 item* 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 Fig. 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.

*2) Requirement Items:* To make the requirements traceable, all the requirements must be specified in the informal specification. The requirements are divided into explicit requirements, and implicit requirements.

*Definition 2:* 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. The 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 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 3:* Let the function $using_D$ be defined as

$$using_D : D_I \rightarrow power(F_I)$$

where $f \in using_D(d)$ iff function item $f$ uses data resource item $d$. Then, the corresponding implicit requirement is defined as

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

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 paper, we use the symbol $power(a)$ to denote the power set of set $a$ where $a$ may be specialized to different sets in different definitions.

The other two kinds 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 4:* Let the two functions $applyingto_D$ and $applyingto_F$ be defined as

$$applyingto_D : C_I \rightarrow power(D_I), \text{ and}$$
$$applyingto_F : C_I \rightarrow power(F_I).$$

Then, the two types of implicit requirements are defined as

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

*3) Inspection Targets:* In an inspection, the inspector is required to examine a specific part of the specification in each step of the animation. We call each specific content that needs to be inspected 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 scenarios, state variables, type declarations, and invariants that are defined in a module. 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 a specific behavior in terms of input and output relation. Because the concept of operation scenario is well defined in our previous publication [45], we do not introduce it in detail here for brevity. A state variable corresponds to a data store in the associated CDFD. 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 is the dependency relations between the formal specification items. Because different items defined in the formal specification work together to present the user's requirements, their dependency 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. As described in Section V, we have developed a prototype tool to support the use of the inspection targets in forming the questions in the checklist.

*Definition 5:* Let $S_F = (M_1, M_2, \ldots, M_n)$ denote a formal specification, where each $M_i (i \in \{1, 2, \ldots, n\})$ 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 $T_M$, $V_M$, $I_M$, and $OS_M$ are the set of all type declarations, state variable declarations, invariants, and operation scenarios, respectively.

Note that, in this 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 condition is part of the definition 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 7:* Let $OS_F = \bigcup_{i=1}^{n} M_i.OS_M$ be the set of all operation scenarios defined in the entire formal specification; let $T_F = \bigcup_{i=1}^{n} M_i.T_M$, and $V_F = \bigcup_{i=1}^{n} M_i.V_M$ 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 targets.

$$IT_1 = OS_F$$
$$IT_2 = V_F$$
$$IT_3 = T_F$$
$$IT_4 = I_F$$

The second class of inspection targets is dependency relation. Inspecting dependency relations aims to check whether formal specification items are used accurately according to the requirements. The basic inspection target operation scenario is directly involved in two kinds of relations. One is between an operation scenario and the state variables that are accessed by the operation scenario, and the other is between an operation scenario and the types of those variables used in the operation scenario. In addition, because a state variable must have a type, a dependency relation also exists between state variables and types.

*Definition 8:* Let the following three functions present the dependency relations between operation scenarios and state vari-

ables, 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 dependency relations are defined as

$$IT_5 = \{(v, os) | v \in V_F \wedge os \in OS_F \wedge os \in using_V(v)\}$$
$$IT_6 = \{(v, t) | v \in V_F \wedge t \in T_F \wedge t = typeof_V(v)\}$$
$$IT_7 = \{(os, t) | os \in OS_F \wedge t \in T_F \wedge t \in typeof_{OS}(os)\} .$$

The function $using_V$ indicates that a state variable can be accessed by more than one operation scenario; 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 relating to more than one type declaration. Each dependency relation is an inspection target, expressed as a pair of relevant items.

Because each invariant is a predicate expressing a property of a type, there should exist a dependency relation between invariants and types.

*Definition 9:* Let $applying_I$ denote the dependency relation between invariants and types.

$$applying_I : I_F \rightarrow T_F$$

Then, the inspection target that corresponds to the dependency relation is defined as

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

Because each type declaration is involved in the dependency relation with operation scenarios or state variables, the invariants can be connected to operation scenarios or state variables through the type declaration.

*Definition 10:* Let the following functions present the dependency 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 dependency relations are defined as

$$IT_9 = \{(i, v) | i \in I_F \wedge v \in V_F \wedge v \in applyingto_V(i)\} ,$$
$$IT_{10} = \{(i, os) | i \in I_F \wedge os \in OS_F \wedge os \in applyingto_{OS}(i)\} .$$

*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 these connections 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.

TABLE II
RELATION 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 general corresponding relations between items in the informal and formal specifications are summarized in Table II. 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 Fig. 1, and the formal specification in Figs. 3 and 4 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 $\{withdraw\_com\}[Receive\_Command,\ Check\_Password,\ Withdraw]\{cash\}$ instead. Because 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.

In the inspection, a trace link should link the operation scenario to the corresponding function items. The function $link_1$ in the first row of Table III formally describes this kind of trace link. In the 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 requirement is the data resource, which is usually realized by a state variable in the formal specification. In our inspection approach, each state variable and its type are linked to a data resource. This linkage is needed because each state variable must be declared with a specific type, and the inspector has to check whether the variable and its type satisfy the original requirement. The function $link_2$ in the table represents the trace link from a state variable and its type to a data resource.

The third type of explicit requirement is a constraint. In practice, a constraint is realized either by an invariant, or by part of a process. The constraints on data resources are generally realized by invariants, and the constraints on functions are usually realized by processes. The functions $link_3$ and $link_4$ in the table 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 dependency relations. The remaining three functions in the table describe the trace links from different dependency relations to corresponding implicit requirements.

Note that not all of the inspection targets can be directly linked to a requirement item. For example, the type declaration

TABLE III
TRACE LINKS

| ID | Inspection Target (Description) | Requirement (Description) | Definition |
|---|---|---|---|
| link1 | IT1 (operation scenario) | RQ1 (function) | $link_1 : IT_1 \rightarrow power(RQ_1)$ |
| link2 | IT6 ((state variable, type)) | RQ2 (data resource) | $link_2 : IT_6 \rightarrow RQ_2$ |
| link3 | IT4 (invariant) | RQ3 (constraint) | $link_3 : IT_4 \rightarrow RQ_3$ |
| link4 | IT1 (operation scenario) | RQ3 (constraint) | $link_4 : IT_1 \rightarrow RQ_3$ |
| link5 | IT5 ((state variable, operation scenario)) | RQ4 ((data resource, function)) | $link_5 : IT_5 \rightarrow RQ_4$ |
| link6 | IT9 ((invariant, state variable)) | RQ5 ((constraint, data resource)) | $link_6 : IT_9 \rightarrow RQ_5$ |
| link7 | IT10 ((invariant, operation scenario)) | RQ6 ((constraint, function)) | $link_7 : IT_{10} \rightarrow RQ_6$ |

is not linked to any of the requirements. However, the dependency 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. When inspecting such dependency relations, the type declaration itself 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.

### B. Properties

To inspect a formal specification, the inspector should also examine four properties: *necessity*, *appropriateness*, *correctness*, and *completeness*.

*1) Necessity:* Checking necessity ensures that all the items defined in the formal specification are necessary. As explained previously, every item defined in the formal specification should contribute to the realization of some user's requirements. Therefore, a formal specification item should not be defined if it is not used anywhere in the specification. For example, a declared type identifier should be used in the declaration of a variable in the specification. If a type identifier is never used, it may imply the existence of defects. This property is defined below.

*Definition 11:* **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) $\forall_{i\in I_F} \exists_{t\in T_F} \cdot t = applying_I(i)$

This property may be automatically checked with tool support. But 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.

*2) Appropriateness:* Appropriateness requires that the inspection targets realize the original requirements in an appropriate manner. Because 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. 4 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 state variable type is appropriate for the data resource.

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

*Definition 12:* **Property 2** If the dependency 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 dependency relations of inspection targets $IT_9$ and $IT_{10}$ should satisfy the following two properties.

*Definition 13:* **Property 3** If the dependency 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 14:* **Property 4** If the dependency relation of $IT_{10}$ is appropriate, the following condition must hold.

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

*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 15 and 16 below.

*Definition 15:* **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 the guard condition, and $D_i$ is the 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 sets 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

TABLE IV
CHECKLIST FOR INSPECTION

| Items | Questions |
|---|---|
| type declaration | Does this type declaration define a type for any data resource?<br>What is the data resource? |
| variable declaration | Does this state variable realize any data resource?<br>What is the data resource?<br>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?<br>Does the function use any data resource?<br>Does this operation scenario use the variable that formalizes the data resource?<br>Does the function comply with any constraint?<br>Does the operation scenario satisfy the invariant that formalizes the constraint or does this operation scenario realize the constraint itself? |

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 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 16:* **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) $^{\sim}P_{pre}(v) \wedge C_i(v) \wedge D_i(v) \Rightarrow Q(t, w)[v/t]$

Because 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 $^{\sim}P_{pre}(v) \wedge C_i(v) \wedge D_i(v)$. Specifically, it requires two effects. 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.

*3) Completeness:* Checking completeness reveals whether all the user's requirements are realized completely. Almost all the requirements are considered to be realized completely if all the related inspection targets are considered appropriate and correct except for function items in the informal specification. According to the definition of the trace link 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. 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 correct form to organize the related operation scenarios of a specific function.

$$\left(os_1^1 \wedge os_2^1 \wedge \ldots \wedge os_x^1\right) \vee \left(os_1^2 \wedge os_2^2 \wedge \ldots \wedge os_y^2\right) \vee \ldots \vee \left(os_1^m \wedge os_2^m \wedge \ldots \wedge os_z^m\right)$$

In the disjunction, all of the operation scenarios, $os_q^p$, link to the same function item in the requirements, and the operation scenarios in a conjunction belong to the same system scenario. By inspecting the disjunction as a whole, the inspector can judge whether a function item is realized completely.

### C. Checklist and Inspection Procedure

In our inspection approach, specification animation is adopted as a reading technique to guide the inspector to read the entire formal specification. As shown in Fig. 5, the inspector can focus on a group of specific inspection targets in each step of animation. The inspector checks the inspection targets by answering related questions in the checklist from Table IV. As mentioned previously, the questions are raised from five aspects, and the checklist can be automatically formed with tool support. One aspect is based on the traceability, and the others are based on the properties of the formal specification. 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 requirement items and inspection targets. Table IV lists the typical questions for each explicitly defined inspection target.

When the inspector reads through the formal specification by following the animation steps, the basic inspection target to be checked is the operation scenario. The operation scenario will lead the inspector to checking other related inspection targets as shown in Fig. 6. In addition to the questions related to the traceability, the inspector also needs to answer the questions raised based on the properties of the specification to ensure that every property is satisfied. For example, according to the necessity property, the inspector should answer the question of whether is it necessary, for every explicitly defined formal specification item. If an item is used somewhere else in the formal specification, then the item is necessary. Otherwise, the item is not necessary. The inspector needs to modify and refine the formal specification based on the answers, as explained in the next section.

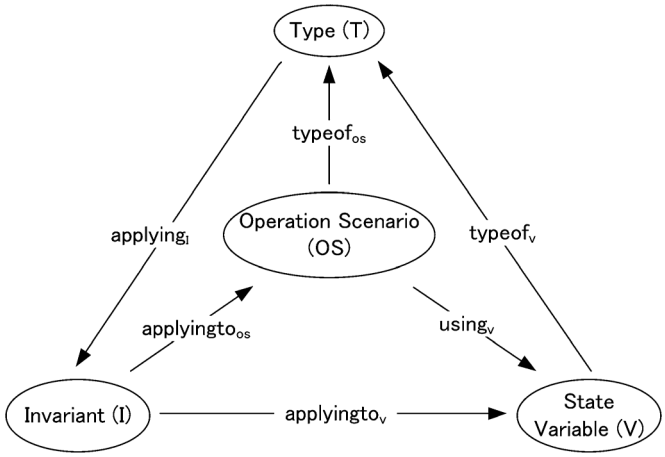


Fig. 6. The dependency relations.

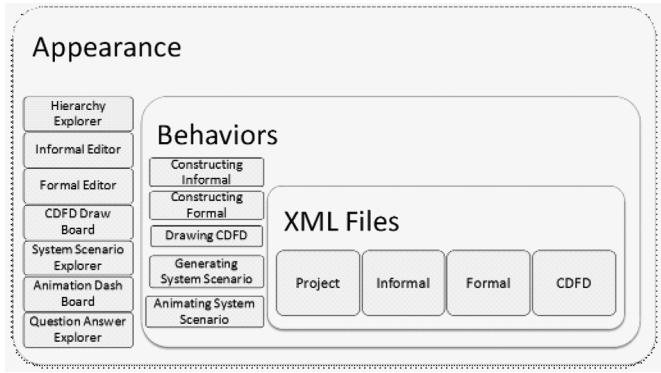


Fig. 7. The architecture of the prototype tool.

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

## V. Supporting Tool

In this section, we present a prototype software tool we have developed over the last five years to support the entire SOFL specification construction and animation-based inspection of formal specifications. As shown in Fig. 7, the architecture of the tool can be divided into three levels. The XML Files level is where the specifications are saved. In our tool, all the specifications, including informal and formal specifications as well as CDFDs, are saved in XML files. The hierarchy of the specifications is stored in a separate XML file with suffix.sofl-project.

The Behaviors level includes two categories of functions for manipulating the XML files. The first category contains the functions used to write specifications, 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*. An explorer is a small window in our tool. Each explorer corresponds to a group of specific functions. For example, the interface for specifying informal specification shown in Fig. 8 includes two explorers. The Hierarchy Explorer on the left-hand side is used to display the hierarchy of the specification, and the Informal Editor on the right-hand side is for specifying informal specification. All the explorers constitute the Appearance level of the tool.

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. In the module, the most important part is process specifications. The tool provides a significant function that always keeps the consistency between the CDFD and the module. When the CDFD is changed, the module will also be properly updated automatically. Fig. 9 shows a snapshot of using the formal specification editor.

After a CDFD is drawn, and all the relevant processes are defined in the associated module, the tool can derive all possible system functional scenarios based on the topology of the CDFD automatically, as shown in Fig. 10. 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.

The inspector can select one of the derived system scenarios for inspection. For example, if the scenario $\{withdraw\_comm\text{-}\}[Receive\_Command,\ Check\_Password,\ Withdraw]\{cash\}$ is selected, and the Animation item in the context menu is chosen, then the interface as shown in Fig. 11 will be displayed. At the top of the window is the Animation Dash Board, which contains several buttons for controlling the animation process. It also provides a spreadsheet for displaying all of the related input and output variables, which are automatically extracted from the specification, and their corresponding values that are supplied by the user.

In each step of the animation, the input data flow, output data flow, and graphical representation of the process involved in the animation will be highlighted. Meanwhile, by using the evaluator provided in the tool, the target operation scenario can be evaluated based on the values for the input data flow and output

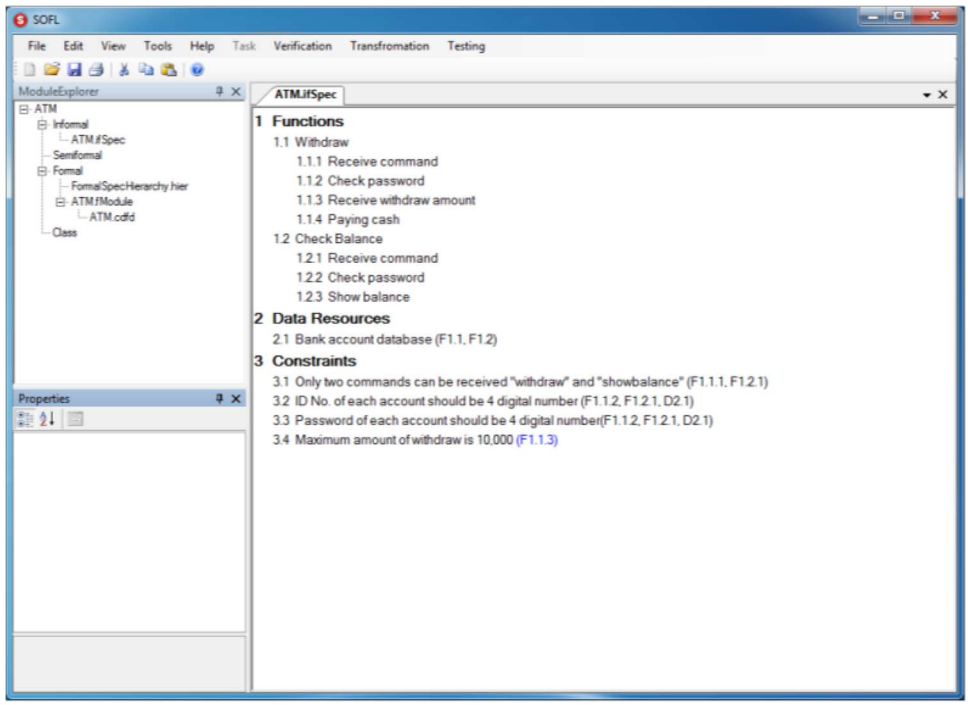


Fig. 8. Snapshot of the informal specification editor.

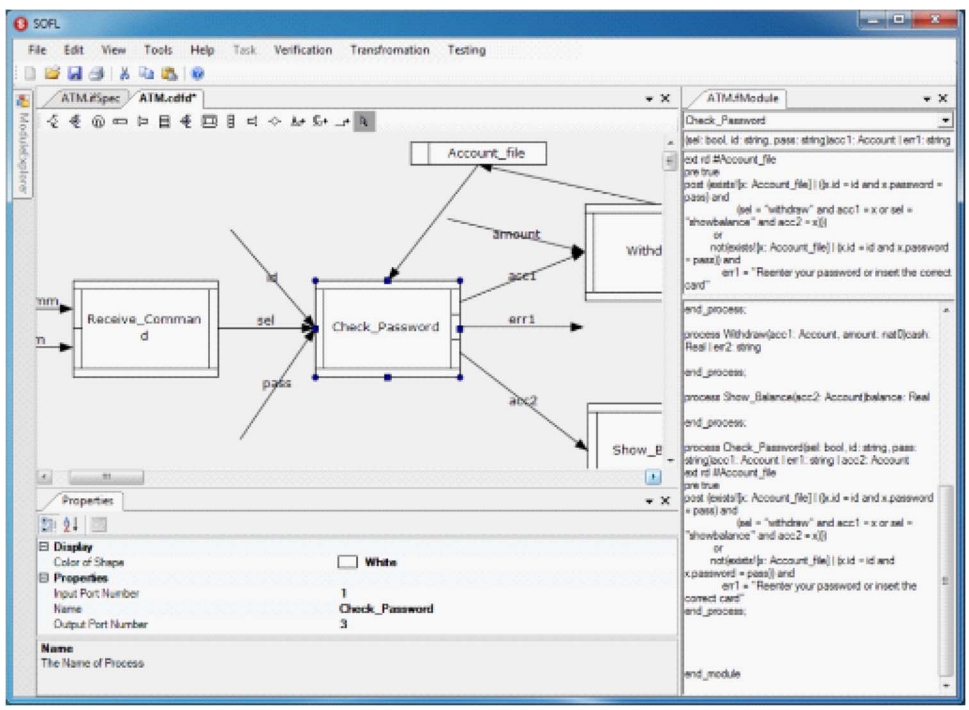


Fig. 9. Snapshot of the formal specification editor.

data flow variables. If the operation scenario is evaluated to true, then the move from the input data flow to the output data flow will be demonstrated; otherwise, it implies the possibility of the values being inappropriate, or the operation scenario containing defects. If the animation can be done, then a window known as the Question Explorer will pop up to display necessary questions to facilitate the inspection of the operation scenario in an interactive manner. After the inspection is completed, the animation will proceed to the next operation scenario occurring in the current system functional scenario for the same procedure. However, if the values for variables or the operation scenario is found to be inappropriate, then the values and the operation scenario will be reviewed for necessary improvement.

Over one hundred students have used the functions of the informal and formal specification editors to build specifications for many applications. This practice allowed us to improve the quality of the tool significantly in both functions and the GUI structure. Currently, our tool can facilitate the designer to easily

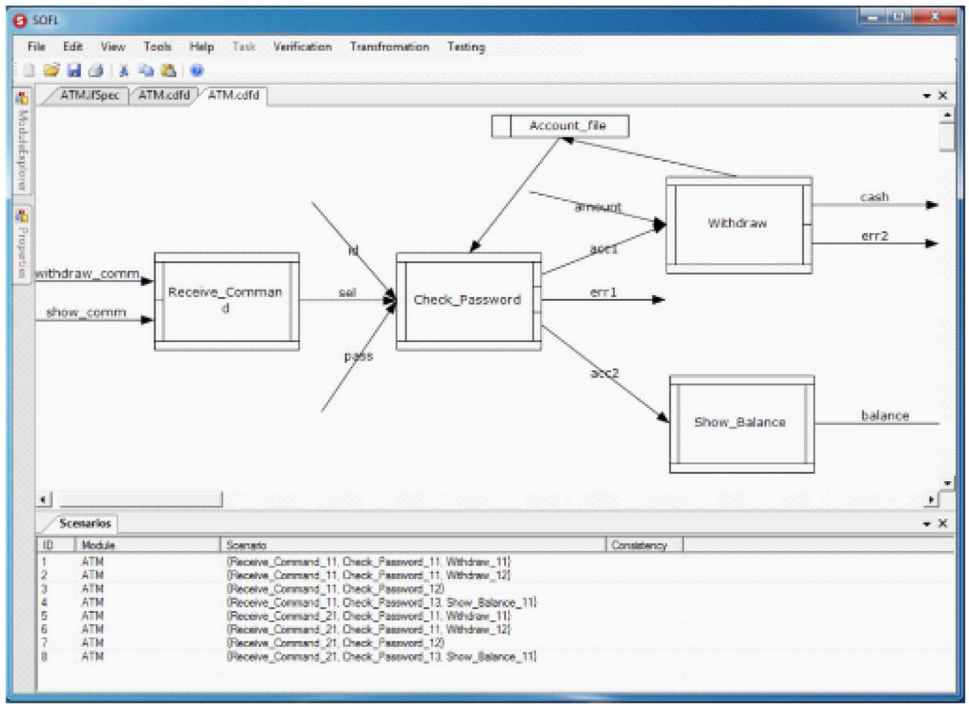


Fig. 10. Snapshot of generating system scenarios.

construct SOFL informal and formal specifications, automatically derive all possible system scenarios from a CDFD, and support all of the important activities of the animation-based inspection technique. In addition to the students' use, our case study to be presented in the next section has also shown that the tool is of high quality, stable, and efficient in supporting both specification construction and specification animation-based inspection.

## VI. 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 rather than on its systematic evaluation.

We choose the ATM software system as the target system for this case study. The entire case study is too large to be presented comprehensibly in this paper. We therefore choose only a part of it for presentation that is sufficient to allow us to explain all of the major aspects of interest. Figs. 3 and 4 show the selected formal specification in our case study. The corresponding requirements are documented in the informal specification, as shown in Fig. 1.

To inspect the formal specification, we use the supporting tool introduced previously to build a checklist that contains questions raised 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 properties 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 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 V 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 11 that describes the relation between the items in the first and third column. 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. For the sake of space, we only demonstrate the process of inspecting one scenario; the inspection of others 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

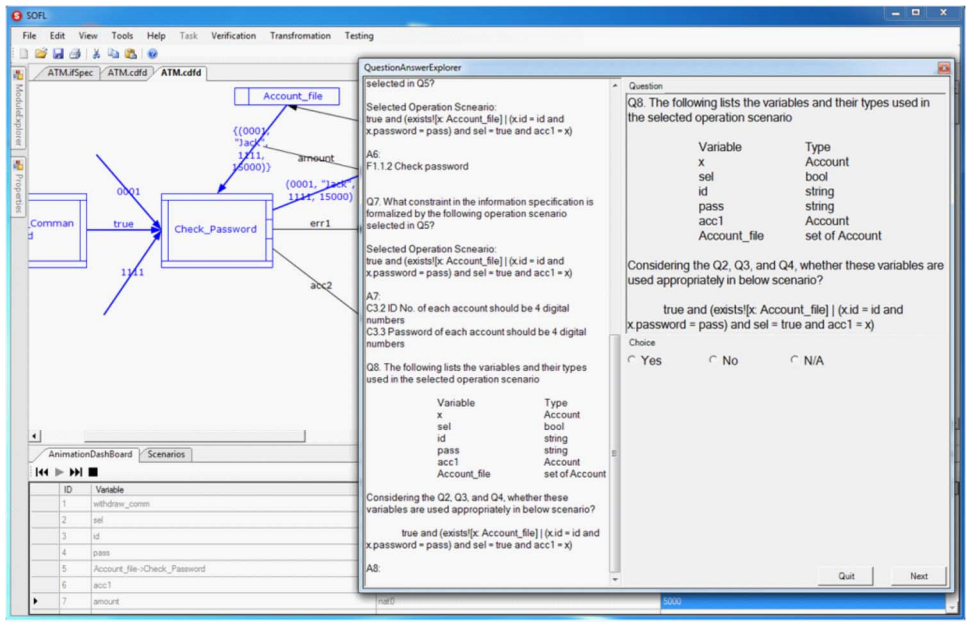


Fig. 11. Snapshot of inspecting a system scenario.

TABLE V
THE RESULT 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 |

TABLE VI
OPERATION SCENARIOS OF THE RELATED PROCESSES INVOLVED IN THE SYSTEM SCENARIO

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

each process that are given in Table VI are extracted for animation. Because 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 VII displays questions for inspecting the selected system scenarios. Because 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.

All 27 questions in Tables VII and VIII are asked in 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 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.
2. The constraint *C3.4* is not formalized in the formal specification.

TABLE VII
INSPECTION RESULT OF THE 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 acc*1 = *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 variables *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*] \| *len*(*a.id*) = 4 |
| 12 | Does invariant *forall*[*a* : *Account*] \| *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 |
| 17 | Is invariant *forall*[*a* : *Account*] \| *len*(*a.id*) = 4 defined complying with SOFL? | Yes |
| 18 | Does invariant *forall*[*a* : *Account*] \| *len*(*a.id*) = 4 appropriately realize the constraint *C3.2*? | Yes |
| 19 | Is invariant *forall*[*a* : *Account*] \| *len*(*a.id*) = 4 used appropriately? | Yes |
| 20 | Is the operation scenario specified complying with SOFL? | Yes |
| 21 | Can precondition *true* implies guard condition (*exists*![*x* : *Account_file*] \| *x.id* = *id and x.password* = *pass*) *and sel* = *true*? | Yes |
| 22 | Can pre and guard condition *true and* (*exists*![*x* : *Account_file*] \| *x.id* = *id and x.password* = *pass*) *and sel* = *true* implyvdefining condition *acc*1 = *x*? | Yes |

TABLE VIII
INSPECTION RESULT OF THE SYSTEM SCENARIO (CONTINUE)

| NO | Question | Answer |
|---|---|---|
| 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 data resource? | Yes, *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 constraint or does this operation scenario realize the constraint itself? | Yes |

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 be devoted 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 done using the *rigorous inspection method* established in

our past publication [46]. Because this method is already made available in the literature, we omit further discussion here for brevity.

## VII. Related Work

After extensively studying the literature, we find that several techniques have been developed for animating or dynamically presenting formal specifications, but no work has explored the same approach as we propose in this paper 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 supports the process of model checking or executing formal specifications. ProB for B-Method is a validation toolset that automatically checks the consistency of B specifications via model checking, and graphically displays the path of state transitions in a counter-example when a violation of the invariant concerned is discovered [47], [48]. An animation starts from the initial state, and proceeds to the next state based on the user's decision. Another tool that adopts model checking for presenting the dynamic behavior of systems is UPPAAL [49], [50], which allows the user to model the system behavior in terms of states, and transitions between states. The simulator of UPPAAL can explore the state space of the model in a step-by-step fashion. Riccobene proposed an automatically driven approach to animating formal specifications in Parnas' SCR tabular notation [51]. 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. VDMTools is an industry-strength toolset supporting the analysis of system models expressed in VDM [52], and has been successfully used in several industrial projects [53], [54]. A large executable subset of VDM can be executed in VDMTools; and the 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 create an external log file recording all the events that happened in an execution. The time tag of each event is used to graphically display all the events on a time-axis. Overture [55] provides functions similar to VDMTools, and is built on an open, extensible platform based on the Eclipse framework. Using the combinatorial testing technique [56], a set of test cases can be generated and used to detect errors such as missing pre-condition, violation of invariant, or violation of post-condition. Prototype Verification System (PVS) [57] is a specification and verification system including an expressive specification language and interactive theorem prover. It offers a ground evaluator that can translate an executable subset of PVS to Lisp [58]. And PVS specifications can be animated by displaying the results of the ground evaluator in the Graphical User Interfaces (GUIs) created by Tcl/Tk [59].

There are also studies on specification animation based on Z notation or other specification languages. PiZA [60] is an animator for Z formal specification. It translates Z specifications into Prolog to generate outputs. Morrey *et al.* developed a tool called wiZe to support the construction of model-based specifications in Z, and the animation of an executable subset of Z notation [61]. The subtool for specification animation 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 it passes the executable representation to ZAL. ZAL animates the specification by executing the specification with test cases. An animation approach for Object-Z Specification is described in [62], which simply translates the specification to C + + code for execution. Time Miller and Paul Strooper introduced a framework for animating model-based specifications by using testgraphs [63]. The framework provides a testgraph editor for the user to edit testgraphs, and then derive sequences for animation by traversing the testgraph. Liu and Wang introduced an animation tool called SOFL Animator for SOFL specification animation [64]. It provides a syntactic and semantic analysis of a specification. When performing an animation, the tool will automatically translate the SOFL specification into a Java program, and then use some test cases to execute the program. To provide the reviewer a graphical presentation of the animation, SOFL Animator uses a Message Sequence Chart (MSC) to present the animation of the operational behaviors. MSC is also adopted in another 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 [65]. 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 [66]. The tool is named ANGOR, and it offers an environment based on a flexible architecture. It allows animating different animation sources, such as formal and executable languages like SDL, and scenario languages like MSC.

Some of the existing studies, such as PVS ground evaluator, wiZe, and SOFL Animator, require an automatic translation from the formal notation into an executable programming language. This translation 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 deal with a subset of the formal notations mentioned above. In contrast to this situation, our animation approach described in this paper does not require any translation of the specifications 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 approach is much easier to implement technically, and is 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 the explicit specifications with test cases. The animation tools based on model checking, such as ProB and UPPAAL, suffer from the inherent challenge of state explosion. Although our animation approach has similarities with some exiting studies, our work differs from the existing work by utilizing animation as a reading technique to support the inspection of formal specifications. Our approach can in principle be applied to any kind of model-based formal specification, but the specific animation-based inspection technique described in this paper is limited to the SOFL CDFD and module notation. The technique is also unlikely to be applicable to non-functional

requirements unless they can somehow be represented in the SOFL formal notation.

As far as the inspection of formal specifications is concerned, there are only a few reports. Guido *et al.* [67] introduce an automatic approach to validating pre-post style 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. In the traditional checklist-based reading techniques, a checklist containing questions on the aspects of interest is usually provided for the inspector to effectively help him or her uncover certain kinds of defects [68], but little instruction on how to perform the inspection is given. In our inspection approach, we provide not only the checklist but also the precise instructions for reading the specifications, and carrying out the inspection. An inspection method called combined-reading technique is introduced in [69]. The method categorizes the defects of specification, and proposes a set of corresponding questions. It requires all the specifications to be checked, but it does not indicate a specific technique to easily read through the specifications. Liu *et al.* proposed a rigorous inspection method called RIM in [46]. The focus of this inspection method is the internal consistency of formal specifications. RIM defines a group of consistency properties, and requires that all these properties be satisfied. Compared to the inspection approach put forward in this paper, RIM focuses on the verification of the 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.

## VIII. Conclusions, and Future Work

Formal specification construction can benefit software development projects by enhancing the understanding of both the user's requirements and potential system behaviors, but it faces a challenge in the validation of the constructed specification due to the difficulty in communication between the user and the designer. We have presented a novel animation-based inspection method to support the validation of formal specifications. The method features the utilization of specification animation as a reading technique, and traceability from formal specifications to the corresponding informal specifications, as well as important properties of the formal specification as the foundation to form questions for the checklist used in the inspection. We have developed a prototype software tool to support our inspection method, which is described in detail from the angles of architecture, functionality, GUI structures, and the mechanism for supporting the integration of the animation-based inspection into the process of specification construction. The significance of the tool lies in its capability of supporting an interleaving process of constructing and inspecting formal specifications, which will increase the opportunities for the user to be actively involved in the process of requirements analysis, and the design for the envisaged system. We have conducted a case study developing a formal specification for ATM software using our method supported by our software tool, and used part of the case study as an example to help illustrate various aspects of our method in the paper.

Our future research will mainly focus on two respects. One is to enhance the functionality of the tool to support more activities in relation to specification animation and specification evolution. Especially, we are interested in exploring more techniques for more effective and efficient animations for validation, such as automatic test suite generation, automatic presentation of the potential behaviors of the system in a virtual environment familiar to the user through specification animation, and automated error detection through the animation-based inspection. The other is concerned with the evaluation of our method. We plan to set up a new project to conduct a systematic experiment on the performance of our method using an industrial scale software system. We believe that our experience acquired from the case study presented in this paper will considerably benefit the future project.

## References


[1] G. Booch, *Object-Oriented Analysis and Design With Applications*. Reading, MA, USA: Addison Wesley Longman, 1994.

[2] A. Shalloway and J. R. Trott, *Design Patterns Explained*, 2nd ed. Upper Saddle River, NJ, USA: Pearson Education, 2005.

[3] H. J. Rosenblatt, *Systems Analysis and Design*, 10th ed. Boston, MA, USA: Cengage Learning, 2013.

[4] J. L. Cyrus, J. D. Bledsoe, and P. D. Harry, "Formal specification and structured design in software development," *Hewlett-Packard J.*, vol. 42, no. 5, pp. 51–58, 1991.

[5] P. Coad and E. Yourdon, *Object-Oriented Design*. Englewood Cliffs, NJ, USA: Yourdon Press Computing Series, Prentice Hall, 1991.

[6] B. Henderson-Seller, *A Book of Object-Oriented Knowledge: Object-Oriented Analysis, Design and Implementation: A New Approach to Software Engineering*. Englewood Cliffs, NJ, USA: Prentice Hall, 1992.

[7] A. M. Davis, "Object-oriented requirements to object-oriented design: An easy transition?," *J. Syst. Softw.*, vol. 30, no. 1, pp. 151–159, 1995.

[8] M. Verhoef, P. G. Larsen, and J. Hooman, "Modeling and validating distributed embedded real-time systems with VDM++," in *FM 2006: Formal Methods*. New York, NY, USA: Springer, 2006, pp. 147–162.

[9] P. G. Larsen, J. Fitzgerald, S. Wolff, N. Battle, K. Lausdahl, A. Ribeiro, and K. Pierce, Tutorial for Overture/ VDM + +, 2010, Tech. Rep.

[10] 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," *Int. J. Softw. Tools Technol. Transfer*, vol. 12, no. 6, pp. 447–466, 2010.

[11] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Comput. Surveys*, vol. 28, no. 4, pp. 626–643, 1996.

[12] J. C. Knight, C. L. DeJong, M. S. Gibble, and L. G. Nakano, "Why are formal methods not used more widely?," in *Proc. 4th NASA Langley Formal Methods Workshop*, Hampton, VA, USA, 1997, pp. 1–12.

[13] D. L. Parnas, "Really rethinking formal methods," *Computer*, vol. 43, no. 1, pp. 28–34, 2010.

[14] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surveys*, vol. 41, no. 4, pp. 1–39, 2009.

[15] S. Liu and R. Adams, "Limitations of formal methods and an approach to improvement," in *Proc. 1995 Asia-Pacific Software Engineering Conf. (APSEC'95)*, Brisbane, Australia, Dec. 1995, pp. 498–507, IEEE CSP.

[16] S. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba, "SOFL: A formal engineering methodology for industrial applications," *IEEE Trans. Softw. Eng. (Special Issue on Formal Methods)*, vol. 24, no. 1, pp. 337–344, Jan. 1998.

[17] S. Liu, *Formal Engineering for Industrial Software Development Using the SOFL Method*. New York, NY, USA: Springer-Verlag, 2004, 3-540-20602-7.

[18] S. Owre, J. M. Rushby, N. Shankar, and F. Von Henke, "Formal verification of fault-tolerant architectures: Prolegomena to the design of PVS," *IEEE Trans. Softw. Eng.*, vol. 21, no. 2, pp. 107–125, 1995.

[19] 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 *Proc. 4th IEEE Int. Conf. Engineering of Complex Computer Systems (ICECCS'98)*, Monterey, CA, USA, Aug. 10–14, 1998, pp. 100–114, IEEE CSP.

[20] H. Hussmann, B. Demuth, and F. Finger, "Modular architecture for a toolset supporting OCL," *Sci. Comput. Program.*, vol. 44, no. 1, pp. 51–69, 2002.

[21] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods: Review and analysis," in *Proc. Espoo 2002*, 2002, p. 478, VTT Publications.

[22] S. Liu, M. Shibata, and R. Sat, "Applying SOFL to develop a university information system," in *Proc. 1999 Asia-Pacific Software Engineering Conf. (APSEC'99)*, Takamatsu, Japan, Dec. 6–10, 1999, pp. 404–411, IEEE CSP.

[23] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, 1976.

[24] NASA, Software Formal Inspections Standard, NASA-STD-2202-93, 1993, Tech. Rep.

[25] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, and M. Zelkowitz, "The empirical investigation of perspective-based reading," *Empir. Softw. Eng.*, vol. 2, no. 1, pp. 133–164, 1996.

[26] D. Kelly and T. Shepard, "Task-directed software inspection technique: An experiment and case study," in *Proc. 2000 Conf. Centre for Advanced Studies on Collaborative Research*, Nov. 13–16, 2000, p. 6.

[27] O. Laitenberger, K. E. Emam, and T. G. Harbich, "An internally replicated quasi-experimental comparison of checklist and perspective-based reading of code documents," *IEEE Trans. Softw. Eng.*, vol. 27, no. 5, pp. 387–421, 2001.

[28] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: Using reading techniques to increase software quality," in *Proc. 14th ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages and Applications*, New York, NY, USA, 1999, pp. 47–56, ACM Press.

[29] T. Thelin, P. Runeson, and B. Regnell, "Usage-based reading-an experiment to guide reviewers with use cases," *Inf. Softw. Technol.*, vol. 43, no. 15, pp. 925–938, 2001.

[30] T. Thelin, P. Runeson, and C. Wohlin, "An experimental comparison of usage-based and checklist-based reading," *IEEE Trans. Softw. Eng.*, vol. 29, no. 8, pp. 687–704, Aug. 2003.

[31] C. B. Jones, *Systematic Software Development Using VDM*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1990.

[32] S. Schneider, *B-Method*. New York, NY, USA: Palgrave, 2001.

[33] S. Liu, M. Asuka, K. Komaya, and Y. Nakamura, "Applying SOFL to specify a railway crossing controller for industry," in *Proc. 1998 IEEE Workshop Industrial-Strength Formal Specification Techniques (WIFT'98)*, Boca Raton, FL, USA, Oct. 20–23, 1998, pp. 16–27, IEEE CSP.

[34] C. L. Ling, W. Shen, and D. Kountanis, "Applying SOFL to a generic insulin pump software design," in *Proc. 2nd Int. Workshop Structured Object-Oriented Formal Language and Method, SOFL 2012*, Nov. 2012, pp. 116–132, Springer.

[35] Y. Wang and H. Chen, "Extension on transactional remote services in SOFL," in *Proc. 2nd Int. Workshop Structured Object-Oriented Formal Language and Method, SOFL 2012*, Nov. 2012, pp. 133–147, Springer.

[36] C. Ho-Stuart and S. Liu, "A formal operational semantics for SOFL," in *Proc. 1997 Asia-Pacific Software Engineering Conf.*, Hong Kong, Dec. 1997, pp. 52–61, IEEE CSP.

[37] S. Liu, "A formal definition of FRSM and applications," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 8, no. 2, pp. 253–281, 1998.

[38] J. Wang, S. Liu, Y. Qi, and D. Hou, "Developing an insulin pump system using the SOFL method," in *Proc. 14th Asia-Pacific Software Engineering Conf. (APSEC2007)*, Dec. 5–7, 2007, pp. 334–341.

[39] C. Morgan, *Programming from Specifications*. Englewood Cliffs, NJ, USA: Prentice Hall, 1990.

[40] M. Fowler and K. Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2002.

[41] S. Liu and J. S. Dong, "Extending SOFL to support both top-down and bottom-up approaches," in *Proc. 2002 IEEE Int. Conf. Systems, Man, Cybernetics (SMC 2002)*, Hammamet, Tunisia, Oct. 6–9, 2002, IEEE CSP.

[42] S. Liu, "Integrating top-down and scenario-based methods for constructing software specifications," *J. Inf. Softw. Technol.*, vol. 54, no. 11, pp. 1565–1572, Nov. 2009.

[43] M. Li and S. Liu, "Automatically generating functional scenarios from SOFL CDFD for specification inspection," in *Proc. 10th IASTED Int. Conf. Software Engineering*, Feb. 15–17, 2011, pp. 18–25.

[44] S. Liu and S. Nakajima, "A decompositional approach to automatic test case generation based on formal specifications," in *Proc. 4th IEEE Int. Conf. Secure Software Integration and Reliability Improvement (SSIRI 2010)*, Singapore, Jun. 9–11, 2010, pp. 147–155, IEEE CSP.

[45] S. Liu, Y. Chen, F. Nagoay, and J. McDermid, "Formal specification-based inspection for verification of programs," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1100–1122, Sep. 2012.

[46] S. Liu, J. A. McDermid, and Y. Chen, "A rigorous method for inspection of model-based formal specifications," *IEEE Trans. Rel.*, vol. 59, no. 4, pp. 667–684, Dec. 2010.

[47] M. Leuschel and M. Butler, "ProB: A model checker for B," in *FME 2003: Formal Methods*. New York, NY, USA: Springer-Verlag, 2003, pp. 855–874.

[48] M. Leuschel and M. Butler, "ProB: An automated analysis toolset for the B method," *Int. J. Softw. Tools Technol. Transfer*, vol. 10, no. 2, pp. 185–203, Feb. 2008.

[49] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems*. New York, NY, USA: Springer, 2004, pp. 200–236.

[50] F. Vaandrager, "A first introduction to UPPAAL," Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook, 18, 2011.

[51] A. Gargantini and E. Riccobene, "Automatic model driven animation of SCR specifications," in *Proc. 6th Int. Conf. Fundamental Approaches to Software Engineering (FASE 2003), Held as Part of the Joint Eur. Conf. Theory and Practice of Software (ETAPS 2003)*, Warsaw, Poland, Apr. 7–11, 2003, pp. 294–309, Springer-Verlag.

[52] J. Fitzgerald, P. G. Larsen, and S. Sahara, "VDMTools: Advances in support for formal modeling in VDM," *ACM Sigplan Notices*, vol. 43, no. 2, p. 3, 2008.

[53] P. G. Larsen and J. Fitzgerald, Triumphs and Challenges for the Industrial Application of Model-Oriented Formal Methods, CS-TR-999, 2007, Tech. Rep.

[54] T. Kurita, M. Chiba, and Y. Nakatsugawa, "Application of a formal specification language in the development of the 'Mobile FeliCa' IC chip firmware for embedding in mobile phone," in *FM 2008: Formal Methods*. New York, NY, USA: Springer, 2008, pp. 425–429.

[55] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, "The overture initiative integrating tools for VDM," *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 1, pp. 1–6, Jan. 2010.

[56] P. G. Larsen, K. Lausdahl, and N. Battle, "Combinatorial testing for VDM," in *Proc. 8th IEEE Int. Conf. Software Engineering and Formal Methods (SEFM 2010)*, Sep. 2010, pp. 278–285.

[57] S. Owre, J. Rushby, and N. Shankar, "PVS: A prototype verification system," in *Automated Deduction-CADE-11*. New York, NY, USA: Springer, 1992, pp. 748–752.

[58] N. Shankar, Efficiently Executing PVS, Computer Science Laboratory, SRI International, Menlo Park, CA, USA, 1999, Tech. Rep.

[59] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, Evaluating, Testing, Animating PVS Specifications, Computer Science Laboratory, SRI International, Menlo Park, CA, USA, 2001, Tech. Rep.

[60] M. Hewitt, C. O'Halloran, and C. Sennett, "Experiences with PiZA: An animator for Z," in *Proc. 1997 Z User Meeting (ZUM'97)*, 1997, pp. 37–51, Springer-Verlag.

[61] I. Morrey, J. Siddiqi, R. Hibberd, and G. Buckberry, "A toolset to support the construction and animation of formal specifications," *J. Syst. Softw.*, vol. 41, no. 3, pp. 147–160, Jun. 1998.

[62] M. Najafi and H. Haghighi, "An animation approach to develop C + + code from object-Z specifications," in *Proc. 2011 CSI Int. Symp. Computer Science and Software Engineering (CSSE 2011)*, 2011, pp. 9–16, IEEE.

[63] T. Miller and P. Strooper, "A framework and tool support for the systematic testing of model-based specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 4, pp. 409–439, 2003.

[64] S. Liu and H. Wang, "An automated approach to specification animation for validation," *J. Syst. Softw.*, vol. 80, no. 8, pp. 1271–1285, 2007.

[65] B. Stepien and L. Logrippo, "Graphic visualization and animation of LOTOS execution traces," *Comput. Netw.: Int. J. Comput. Telecommun. Netw.*, vol. 40, no. 5, pp. 665–681, 2002.

[66] P. Combes, F. Dubois, and B. Renard, "An open animation tool: Application to telecommunication systems," *Comput. Netw.: Int. J. Comput. Telecommun. Netw.*, vol. 40, no. 5, pp. 599–620, 2002.

[67] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel, "Automated abstractions for contract validation," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 141–162, 2012.

[68] M. Staron, L. Kuzniarz, and C. Thurn, "An empirical assessment of using stereotypes to improve reading techniques in software inspections," in *Proc. 3rd Workshop on Software Quality, 3-WoSQ*, New York, NY, USA, 2005, pp. 1–7, ACM.

[69] A. A. Alshazly, A. M. Elfatatry, and M. S. Abougabal, "Detecting defects in software requirements specification," *Alexandria Eng. J.*, vol. 53, no. 3, pp. 513–527, 2014.

**Mo Li** is a Ph.D. candidate in the Graduate School of Computer and Information Sciences at Hosei University, Japan.

His research interests include software engineering, applied formal methods, and formal specification verification and validation.

**Shaoying Liu** received the Ph.D. in Computer Science from the University of Manchester, U.K. in 1992.

He is Professor of Software Engineering at Hosei University, Japan. His research interests include Formal Engineering Methods for Software Development, Specification Verification and Validation, Specification-Based Program Inspection, Specification-Based Program Testing, and Intelligent Software Engineering Environment. He has published a book titled "Formal Engineering for Industrial Software Development Using the SOFL Method" with Springer-Verlag, eight edited conference proceedings, and over 150 academic papers in refereed journals and international conferences. He proposed to use the terminology of "Formal Engineering Methods" in 1997, has established Formal Engineering Methods as a research area based on his extensive research on SOFL (Structured Object-oriented Formal Language) and its related technologies since 1989, and the development of the ICFEM conference series since 1997. In recent years, he served as General Co-Chair of the International Conference on Formal Engineering Methods (ICFEM 2012), Program Co-Chair of the International Workshop on SOFL+MSVL (SOFL+MSVL 2013 and 2014), Steering Committee Chair for ICFEM 2009~2014, and PC member for numerous international conferences. He is on the editorial board for the *Journal of Software Testing, Verification and Reliability* (STVR). He is a Fellow of British Computer Society, a Senior Member of the IEEE Computer Society, and a member of the Japan Society for Software Science and Technology.