Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1271–1285
The Journal of Systems and Software
www.elsevier.com/locate/jss

An automated approach to specification animation for validation

Shaoying Liu [a,*], Hao Wang [b]

[a] Department of Computer Science, Hosei University, 3-7-2 Kajino-cho, Koganei-shi, Tokyo 184-8584, Japan
[b] Shanghai Jiaotong University, China

Received 14 November 2005; received in revised form 30 November 2006; accepted 7 December 2006
Available online 20 December 2006

Abstract

Formal specification has been increasingly adopted for the development of software systems of the highest integrity. However, the readability of specifications for large-scale and complex systems can be so poor that even the developers may not easily understand whether their specifications define the ''intended behaviors''. In this paper, we describe a software tool that supports the animation of specifications by simulating their functional scenarios using the Message Sequence Chart (MSC). The tool extracts automatically functional scenarios from a specification and generates a message sequence chart for each of them for a syntactic level analysis. The tool can also execute a functional scenario with test cases for a semantic level analysis if all the processes involved in the scenario are defined using explicit specifications. With the tool support the animation of a specification can be carried out incrementally to assist its user to review the adequacy of the specification. We present a case study applying the tool to animate a formal specification for a library system and evaluate its result.

Keywords: Specification animation; Validation; Formal specification

1. Introduction

It is well recognized that starting the development of a software system of the highest integrity by writing its specification in a formal notation is substantially beneficial (Bowen and Hinchey, 1999; Hinchey and Bowen, 1999). Due to the well-defined syntax and semantics of the formal notation, the formal specification provides precise and concise requirements for the proposed software system. In particular, formal specification languages with the feature of integrating textual formal notations with intuitive graphical notations developed in recent years, such as the Structured Object-Oriented Formal Language (SOFL) (Liu et al., 1998c; Liu, 2004b), the combination of Unified Modeling Language (UML) and Object Constraint Language (OCL) (Warmer and Kleppe, 1999), and Cleanroom (Mills et al., 1987; Linger and Trammell, 1999), have even greater potential to be adopted by practitioners for wider applications. Apart from the advantage of forcing developers (the persons who build systems) to clarify ambiguities in requirements by writing formal specifications, the resultant specifications also offer a firm ground for the verification and validation of the formally defined functions and other important properties (e.g., safety, security) under effective and efficient tool support. However, as Leveson points out in (Leveson, 2000), the adoption of formal specifications in industry also faces various challenges. One important challenge is that formal specifications are usually written in a complex structure and are difficult to read and understand. This is partially because large scale systems tend to have a complex functionality and properties, and the formalization of them usually leads to a complex structure of the involved components and the concentration of detailed formal expressions defining the functions of operations. It is also because formal specifications may not intuitively reflect the corresponding concepts and behaviors of systems in the real world. For this reason, it

* Corresponding author.
E-mail address: sliu@k.hosei.ac.jp (S. Liu).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.12.540

is not easy for people to build a mental "bridge" between expressions in formal specifications and perceptions of the corresponding systems in the real world. Of course, the lack of training in formal specification may exacerbate the difficulty in practice.

Since detecting errors in requirements specifications reduces much more cost than detecting errors in programs (Boehm, 1981), it is significantly beneficial to verify and validate specifications before their implementations. Specification animation was developed as an effective technique for this purpose. As Miller and Strooper pointed out in (Strooper and Miller, 2001), animation serves two purposes: (1) giving end users and field experts a chance to interact with the specification and observe its operational behavior, and (2) providing the specifier with concrete examples of how the specification behaves, so that they can check whether the specification reflects the intended properties of their design. Some tools have been built to support animation of different specification languages, such as PiZA (Hewitt et al., 1997), ZAL animation system (Morrey et al., 1998), Possum (Hazel et al., 1997), B-Model animator (Waeselynck and Behnia, 1998), and ANGOR (Combes et al., 2002), but most of those tools focus on the approach similar to testing: executing specifications with sample input values and analyzing the results. In general, this kind of technique requires a translation from a formal specification language to an executable programming language (e.g., Prolog or LISP), which imposes many restrictions to the style of the specifications written in the specification language. It also disallows the user of the tool to analyze the contents of the specifications.

In this paper we propose a different approach to specification animation that facilitates reviews of specifications. Using this approach, a reviewer (the person who conducts the animation) reads through the target specification to digest the contents and to analyze the potential behaviors of the specification in order to detect errors in relation to the validity or consistency of the specification. The major principle underlying the animation approach is to allow the reviewer to check every possible functional scenario defined in the specification. A functional scenario is a sequence of operations that defines a specific kind of behavior. An example of such a scenario is to withdraw money from an Automated Teller Machine (ATM): receiving the request, checking the card and password, and finally providing the requested amount. We have developed a software tool to support this approach. The tool automatically generates all the possible functional scenarios for a given specification. It also provides assistance to the reviewer in animating each single scenario by simulating the potential behavior in terms of transforming an input to an output through the sequential executions of the operations involved in the scenario. As described in detail in Section 4, the tool offers support for both syntactic and semantic level animations, and provides an effective mechanism to facilitate the reviewer to flexibly control the animation process. We have also conducted a case study using the tool for the animation of a library system. Our experience shows that the tool is easy to use and effective in detecting errors, although there are also aspects for further improvement, as described in detail in Section 5.

Since the animation tool needs to support a specific formal notation, we choose the Structured Object-oriented Formal Language (SOFL) (Liu, 2004b; Liu et al., 1998c) as the target language to support for three reasons. One reason is that SOFL integrates the commonly used formal method Vienna Development Method (VDM) (Jones, 1990) with the intuitive data flow diagram (DFD) (Yourdon, 1989). The DFD is used to model the architecture of a system and an extended VDM specification language (VDM-SL) is adopted to formally define all the components involved in the CDFD, such as processes, data flows, and data stores. Since DFD is intuitive and widely applied by practitioners for system analysis and design (Yourdon, 1989) and VDM-SL offers a powerful notation for precisely defining the components of DFD, SOFL specifications are comprehensible at both the architecture level and the component level. This will potentially benefit the communications between the end users and the developers, and even among the developers. This property is especially important for specification animation because the communications between the developers and the end users may be required during the animation process in order to make correct judgements in validating the specification. The second reason is that SOFL has been taught to both undergraduates and graduates over the last seven years, especially in Japan and China. It has been applied to the modelling of computer-controlled safety-critical systems, such as the railway crossing controller in collaboration with Mitsubishi Electric Research Institute (Liu et al., 1998a,b). It has also been applied in the development of network protocols (Taguchi, 2000; Liu, 2003a) and of some commercially-based systems (Liu et al., 1999; Liu, 2003b, 2004a). In addition to its direct applications, SOFL has been used as the base-language for developing aspect-oriented specification paradigm (Shen and Chen, 2005), modelling technology for developing middleware-based transaction management (Chen et al., 2005), and testing techniques (Liu, 1999; Offutt and Liu, 1999). Together with the progress in support tools, SOFL has a great potential for industrial application in the future. The final reason is that our expertise in SOFL and previous experience in research on SOFL *specification review* enable us to have a good insight into the fundamental principle of the proposed animation and to effectively develop the technique and the tool. Although the discussion in this paper is based on the SOFL specification language, the proposed animation technique can be easily extended for animations of specifications in other related languages, such as VDM-SL and DFD.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the SOFL specification language in order to pave the way to introduce our animation approach. Section 3 describes the underlying principle

of the animation approach. Section 4 introduces the functions of our animation tool. In Section 5 we present a case study and in Section 6 we overview the related work. Finally, in Section 7, we conclude the paper and point out future research directions.

2. Brief introduction to SOFL

SOFL is both a language and a method for constructing functional specifications and designs of software systems. The introduction in this section focuses only on the SOFL specification mechanisms serving the purpose of this paper. A detailed introduction to the SOFL technology is described in Liu's recent book (Liu, 2004b).

The essential structures of a SOFL specification are modules and Condition Data Flow Diagrams (CDFDs). A CDFD, which is a DFD with an operational semantics, is a directed graph composed of processes describing functional operations, data flows for data communications among processes, and data stores. Each process is defined with a pair of logical conditions, called pre- and postconditions, imposing constraints on the inputs and outputs before and after an execution of the process, respectively. A process is similar to an operation in VDM, but different in that it may take input data flows and produce output data flows exclusively. This means that a process is allowed to receive several groups of input data flows, but it consumes one of the groups in order to execute the process each time to produce one of the output data flow groups. It is required by SOFL that the corresponding pre- and postconditions of the process need to be given in a disjunction in which each disjunctive clause defines constraints only on one of the input or output group variables. This mechanism enables a process to have a stronger capacity than an operation in VDM for functional abstraction. To achieve well-structured documentation, we put the formal definitions of processes, data flows, and data stores in a *module*. Syntactically, a module offers a structure to define all the components occurring in the associated CDFD, but semantically it represents an operation whose behavior is defined by the associated CDFD and the related process specifications. A comprehensible example helps understand the structure. Fig. 1 shows the CDFD of a simplified ATM and Fig. 2 shows the outline of its associated module.

The module, together with the CDFD, defines an ATM model (of course, there are other kinds of ATM models, such as the one described in Gomaa's book (Gomaa, 2000), but this simple example serves our purpose). Once a customer's request, denoted by data flow withdraw_comm or Balance_comm, arrives at the process Receive_Command, the process will determine the type of the request (for withdrawal or for showing the balance) and the decision is represented by the data flow sel. The data flow then reaches the process Check_Password, and the process will check the customer's card id and password, which are represented by id and pass, respectively, against the account files containing all the customers' accounts, which is represented by the data store Account_file. If the card_id and pass are confirmed to be correct, the account information will be transferred to the process Withdraw through data flow account1 or to the process Show_Balance through data flow account2; otherwise, if the card_id or pass does not match the corresponding id or pass in the data store Account_file, then an error message, represented by data flow e_msg, will be produced. In the case of the process Withdraw, if the amount desired to be withdrawn is greater than that available for withdrawal on the account, an error message warn_msg will be produced; otherwise, the requested amount of cash, represented by data flow cash, is delivered.

As far as the animation is concerned, the structure of process specification is important. The process Check_Password defined below is an example showing the structure of a process specification.

process Check_Password (card_id: **nat0**, sel: **bool**, pass: **nat0**)

account1: Account | e_msg: **string** | account2: Account

ext **rd** Account_file

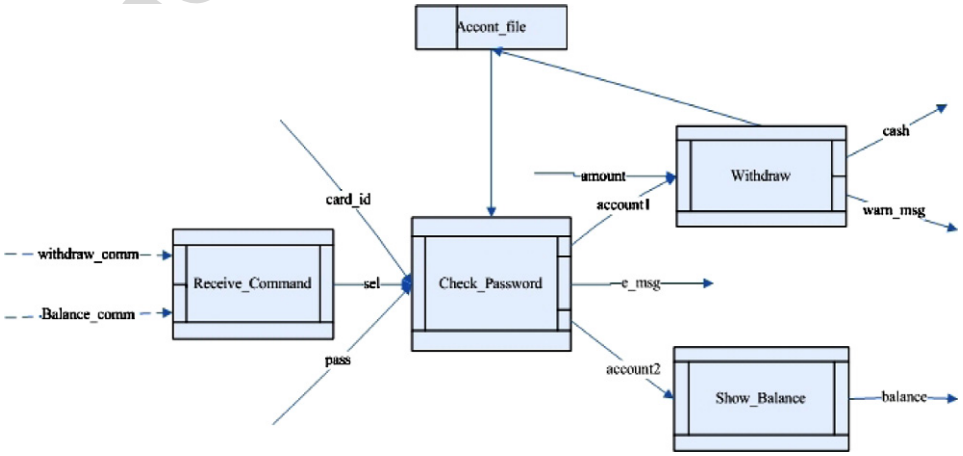


Fig. 1. The CDFD of a simplified ATM.

```
module SYSTEM_ATM;
type
Account = composed of
            id: string
            name: string
            password: nat0
            balance: real
            available_amount: real
            end;
var
 Account_file: set of Account;
inv
 forall[a:Account] | len(a.id)>0;

process ReceiveRequest(req:bool) sel: bool;
pre...
post...
comment...
end_process;
process CheckPassword(id:nat0,sel:bool,pass:nat0) acc1:Account | err1:string | acc2: Account;
pre...
post...
comment...
end_process;
process Withdraw(amount:nat0,acc1:Account) err2: string | cash:nat0
pre...
post...
comment...
end_process;
process ShowBalance(acc2:Account) balance: nat0;
pre...
post...
comment...
end_process
end_module
```

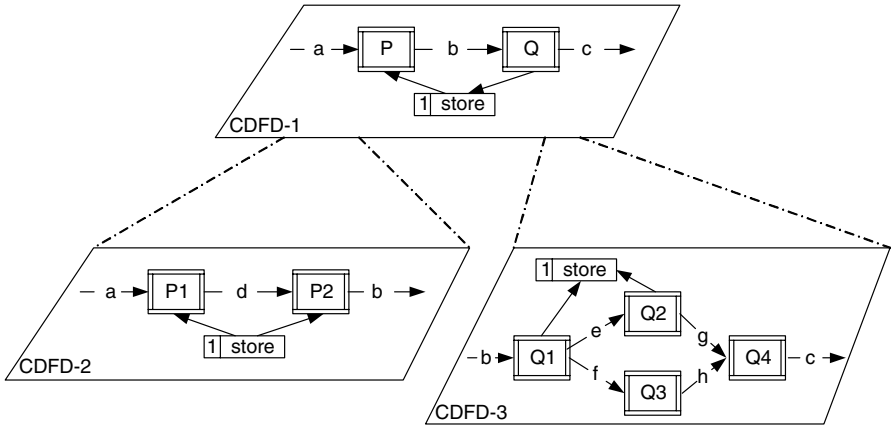
Fig. 2. The module associated with the CDFD for the ATM.

```
pre      true
post     (exists![x: Account_file] | ((x.id = card_id and
         x.password = pass) and
         (seal = false and account1 = x or
          seal = true and account2 = x)))
          or
          not (exists![x: Account_file] | (x.id = card_id and
              x.password = pass)) and e_msg = "Reenter
              your password or insert the correct card"
end_process;
```

where Account is a composite type and Account_file is an external variable (representing a store) as defined in Fig. 2. The precondition of the process is true, indicating that there is no specific constraint on the initial state. The postcondition shows how each of the three output variables account1, e_msg, and account2 is defined based on the input and external variable.

To scale up the modeling technology using SOFL in practice, a specification in large can be constructed as a hierarchy of CDFDs by decomposition of processes. Hierarchical abstraction is an effective way to deal with complexity, as emphasized by Leveson (2000). To achieve encapsulation and information hiding for the evolution of specifications, data flows and stores in CDFDs can be defined as instances (or objects) of some classes. SOFL provides constructs and rules for defining classes and allows the defined classes to be used for the declarations of data flow and/or store variables in process specifications in a module. Thus, the whole structure of a SOFL specification is a hierarchy of CDFDs and their associated modules, surrounded by the specifications of classes that may be associated with each other, as illustrated in Fig. 3. Such a specification has both a clear representation of the system's main functions in the hierarchy of CDFDs and the flexibility of defining necessary components (classes and objects) for the definitions of the main functions.

SOFL can also be used to build specifications in object-oriented manner from the beginning. The way to do it is to concentrate on identifying objects, defining their classes, and building data flow communications among the objects. A CDFD can be used to describe data flow communications among objects if we use each process in the CDFD to represent an object. A process in a CDFD is allowed to have several input ports and output ports, and each input or output port may accommodate a group of input or output data flows. A process in general defines exclusive relations between input ports (and their input data flows) and output ports (and their output data flows). Such a




```
class C1;
const,var declarations;
inv a list of invariants; /*can be empty*/
method Init() /*conductor*/
method M1 specification;
method M2 specification;
end_class;

class C2/C1;
const, var declarations;
inv a list of invariants
method Init()
method H1 specification;
method H2 specification;
end_class;
. . .
class Cn/C2;
const,var declarations;
inv a list of invariants
method Init()
method N1 specification;
method N2 specification;
method N3 specification;
end_class;
```

```
module SYSTEM_Spec;
const, type, var declarations;
inv a list of invariants; /*can be empty*/
behav CDFD_1;
process Init() /*initialization*/
process P specification;
process Q specification;
end_module;

module P_Decom/SYSTEM_Spec;
const, type, var declarations;
inv a list of invariants
behav CDFD_2;
process Init()
process P1 specification;
process P2 specification;
end_module;

module Q_Decom/SYSTEM_Spec;
const, type, var declarations;
inv a list of invariants
behav CDFD_3;
process Init()
process Q1 specification;
process Q2 specification;
process Q3 specification;
process Q4 specification;
end_module;
```

Fig. 3. An illustration of the general structure of a SOFL specification.

process is suitable for representing an object because the relation between each input and output ports (and their data flow groups) provides a specification for the implementation of a method of the object. According to our previous experience (Gomaa et al., 2000), the approach to modeling object-oriented systems based on data flow communications among processes (representing objects) in CDFDs allows us to concentrate on defining "what to be done" by the system, without the need of considering how the data flow communications are implemented as message communications among objects in programs (or their abstract representations in MSC). Such specifications are also intuitive for many clients, and therefore facilitates their validation, before they are actually implemented.

3. The principle of animation

We describe the principle of the animation approach by discussing the *animation strategy* and *animation process*. The strategy reflects what to do in an animation, while the process supplies a procedure for systematically performing the animation.

3.1. Animation strategy

Since the goal of a specification animation is to facilitate the reviewer to understand the potential behaviors of the specification and detect errors by analyzing the potential behaviors, we take the following strategy for specification animation: *analyzing every possible functional scenario defined in the specification*. A functional scenario, or simply a scenario, defines a specific kind of functional behavior of the systems through a sequential executions of operations. A kind of behavior is usually modeled by a relation between the input and output, that is, given an input, the result of a behavior of the system results in a certain output. However, syntactically, the presentation of a scenario may vary for different specification languages. In the SOFL specification language, a scenario is represented by a sequence of processes with a pair of input and output data items, as detailed in Definition 1.

Definition 1. A scenario of a specification is a sequence $d_1[P_1, P_2, \ldots, P_n]d_2$, where each $d_i(i \in \{1,2\})$ is a pair of data flow set v_d and external variable set v_e, i.e., (v_d, v_e), and each $P_j(j \in \{1,2,\ldots,n\})$ is a process. d_1 is called the input data item, while d_2 is called the output data item.

The scenario $d_1[P_1, P_2, \ldots, P_n]d_2$ defines a behavior that transforms input data item d_1 into the output data item d_2 through a sequential execution of processes $P_1, P_2, \ldots, P_n$. In fact, in order to execute each process in the scenario, other input data flows may also be needed and the execution usually yields some output data flows. Since those data flows are used or produced within the process of executing the entire scenario, they are called *intermediate data items.* As all the intermediate data items are attached to the corresponding processes and adding them syntactically in the scenario may complicate the representation of the scenario, we deliberately omit all intermediate data items, but merely keep the input data item of the first process of the scenario as the input data item of the scenario and the output data item of the last process as the output data item of the scenario.

Consider the CDFD in Fig. 1 as an example. It defines the following five scenarios:

1. ({*withdraw_comm*}, {~*Account_file*})[*Receive_Request*, *Check_Password*, *Withdraw*]({*warn_msg*}, {*Account_file*})
2. ({*withdraw_comm*}, {~*Account_file*}) [*Receive_Request*, *Check_Password*, *Withdraw*]({*cash*}, {*Account_file*})
3. ({*withdraw_comm*}, {~*Account_file*})[*Receive_Request*, *Check_Password*]({*e_msg*}, {*Account_file*})
4. ({*Balance_comm*}, {~*Account_file*})[*Receive_Request*, *Check_Password*]({*e_msg*}, {*Account_file*})
5. ({*Balance_comm*}, {~*Account_file*})[*Receive_Request*, *Check_Password*, *Show_Balance*]({*balance*}, {*Account_file*})

where all the intermediate data flows (e.g., *sel*, *id*, *pass*) are omitted in the scenario expressions for brevity, as explained above. Such a functional scenario clearly describes how the final output data flows and values of the stores are produced by a sequence of processes based on the input data flows and the initial values of the stores. For example, the scenario 2 above describes how the data flow *cash* and the final value of the store *Account_file* are produced by the processes *Receive_Request*, *Check_Password*, and *Withdraw* based on the input data flow *withdraw_comm* and the initial value of the store *Account_file* (denoted by ~*Account_file*). An algorithm for generating all the scenarios in a CDFD was introduced in our previous publication (Liu et al., 1998d) and our animation tool is implemented on the basis of the algorithm, as described in Section 4.2.2.

3.2. Animation process

In order to efficiently support our animation strategy, we take the following process to animate a specification:

Step 1. Derive all the possible scenarios and the related pre- and postconditions for each processes in each scenario from the specification.
Step 2. For a selected scenario, generate a Message Sequence Chart (MSC) to represent it graphically, and analyze the potential behavior of the scenario by "dynamically" generating the data flows between processes in the scenario.
Step 3. Execute the scenario with test cases (input values for the first process in the scenario).
Step 4. Repeat Step 2 to Step 3 until all the scenarios are exhausted.

Our animation strategy requires that every scenario of the specification be animated, therefore, the first step of an animation is to derive all the possible scenarios from the specification. Since the related pre- and postconditions of each process in the scenario are used to produce test cases for the execution of the scenario in Step 3, they are also required to be derived. In the second step above, a scenario is selected for animation, either based on the user's request or otherwise at random. Since MSC described in Haugen (2001) is an intuitive and commonly used notation for representing interactions between system components through message communications, we adopt it to represent the animation of a scenario. Our purpose of adopting MSC is however not to show how data flow communications among processes are implemented as message communications among objects, but to help the reviewer focus on the analysis of how data items flow among processes in a selected scenario. To this end, we treat processes as instances (objects) and data flows as messages in the MSC notation. The tool we have built to support the animation process, as introduced in Section 4, facilitates the reviewer to perform *syntactic level analysis* of the behavior of the scenario by allowing him or her to flexibly control the (syntactic) flows of data items from one process to the next adjacent one in the scenario. For this purpose, the tool offers the monitoring operations, including "start", "pause", "forward", and "stop"; each of the operation can be applied by clicking on the corresponding button, see Section 4 for details. Apart from the syntactic level analysis, Step 3 requires a semantic level analysis by executing the scenario with test cases.

The semantic level analysis concentrates on checking the consistency and the validity of a scenario based on the semantics of both the formal and informal specification of each process involved. The formal specification of a process is mainly used to check the internal consistency of the scenario, while the informal specification, which is also in the form of informally expressed pre- and postconditions, is used to check the validity of the scenario. To carry out a semantic level analysis, some test cases for the input data flows of the scenario are first generated to satisfy the preconditions of the corresponding starting processes of the scenario. The tool will then automatically produce the output data flows of the processes involved in the scenario

based on their formal specifications. With those test cases generated by the reviewer and derived by the tool, the reviewer is allowed to examine whether the input data flows can be transformed into the output data flows of the scenario. When executing a process during the semantic level analysis of the scenario, the reviewer checks the validity of the process by comparing the test cases and their corresponding execution results to the informal specification of the process. The reviewer needs to judge when the test cases satisfy the precondition of the process, whether their execution results also satisfy its postcondition. Since both the pre- and postconditions of the informal process specification are informal expressions (e.g., English), the judgement is not made by the software tool but by the reviewer.

In general, it is important to animate all the scenarios derived from the specification in order to ensure that all the specified behaviors are analyzed, unless the reviewer requests to animate only those he or she believes important for the validation of the specification. Considering the cost and time incurred from animating all the scenarios, we believe that conducting a *selective animation* (i.e., selecting only important scenarios for animation) based on the reviewer's engineering judgement is an effective approach in practice.

4. An animation tool

We have built a prototype software tool, called SOFL-Animator, to support the animation approach described previously. In this section, we introduce the tool by explaining its major functions and discussing the important implementation issues.

4.1. Functions

The SOFL-Animator supports all the activities of the entire animation process described in the previous section. Specifically, it offers the following major functions:

- *Automatically deriving all possible scenarios from SOFL specifications*. The tool implements the algorithm extended based on the one described in our previous publication (Liu et al., 1998d) for generating all possible scenarios from one level CDFD and its corresponding module of a SOFL specification. As demonstrated by the example of generating the scenarios from an ATM specification in Section 3.1, the specific idea of the algorithm is first to determine all the possible input data flow groups of the CDFD, where each group is capable of executing a scenario. Then, starting from each input data flow group, the algorithm searches the scenario that transforms the input data flow group into an output data flow group on the basis of the CDFD structure. Finally, the related stores are added to both the input data flow group and the output data flow group of each scenario to explicitly indicate what input data flows and stores are transformed into what output data flows and stores.
- *Managing scenarios.* All the scenarios generated from the specification are named, and their names are organized as a list and shown in the GUI of the tool for selection by the reviewer. Upon the selection of a scenario name on the list, the tool will automatically highlight all the related data flows, stores, and processes of the corresponding scenario in the CDFD to draw attention from the reviewer. The tool also allows the reviewer to rename or eliminate a scenario for whatever reason. For example, a name "path_1" denoting a scenario generated by the tool can be changed to "Withdraw_money" for comprehensibility.
- *Automatically generating MSC for a scenario.* The tool treats processes in the scenario as instances and data flows as messages in the MSC generated. Since MSC is an intuitive representation of the execution sequence and interaction between message-passing instances (or components), animation of a scenario in the MSC format will be so comprehensible that the reviewer will be able to easily and accurately make judgements in understanding the animated behaviors and in detecting errors.
- *Supporting the syntactic level analysis of a scenario.* The tool supports both syntactic level analysis and semantic level analysis. In the syntactic level analysis, the reviewer is allowed to exam how the input data flows are used to activate the scenario syntactically to produce the output data flows through the transformations performed by the processes involved. This kind of analysis is performed entirely based on the structure of the related CDFD; it does not use the contents of process specifications.
- *Supporting the semantic level analysis of a scenario*. In this kind of analysis, not only are data flow transformations examined syntactically, but they are analyzed semantically by considering the effect of the mathematical definitions in the pre- and postconditions of the processes in the scenario. The way to perform such an analysis is first to generate test cases as input data flows of the scenario and then to execute the scenario by interpreting the pre- and postconditions of the processes, provided that they are written in the format that only equation is used to define output data flows and stores. Our tool supports all the related activities in this process, including test case generation and interpretation of process specifications.
- *Monitoring the status of entity.* This function is aimed at supplying the current information of the current entities (e.g., variables, processes) after every transformation step during the animation of a scenario. The purpose is to facilitate the reviewer to check both the data and operation aspects of the entities of the scenario. The status of a variable is reflected by its current value and the status of a process can be one of the three states: "normal", "enabled", and "executing". The status "normal" indicates that the process is neither enabled nor executing; "enabled" indicates that the process is ready to

execute because of the availability of the required input data flows; and "executing" means that the process is in execution.

4.2. The implementation

We implemented the animation tool using Java under the Eclipse environment in the Highly Reliable Software Laboratory at Shanghai Jiaotong University. The tool also depends on the *SOFL Specification Constructor* developed previously by our group (Xue, 2005) in that the target CDFD for animation is supplied by the specification constructor. In this subsection, we discuss the implementation issues in relation to the structure of the animation tool, scenario generation, MSC generation, operations of MSC, and execution of scenarios.

4.2.1. The overall structure

Fig. 4 shows the overall structure of our tool. The SOFL-Animator takes a specification from the SOFL Specification Constructor and generates all the possible scenarios by the Functional Scenarios Generator. The scenarios generated are then used to provide necessary data to the MSC Generator, Animation Controller, and Status Monitor. For a selected scenario, the MSC Generator produces a MSC for the animation of the scenario. The Animation Controller receives the information of the selected scenario and supports the reviewer to perform the animation of the scenario within the MSC framework, including both syntactic and semantic level analyses. It also highlights the corresponding scenario in the CDFD of the specification upon the request from the reviewer. The Status Monitor records the state of every animation step and displays them in an appropriate format in the animation GUI.

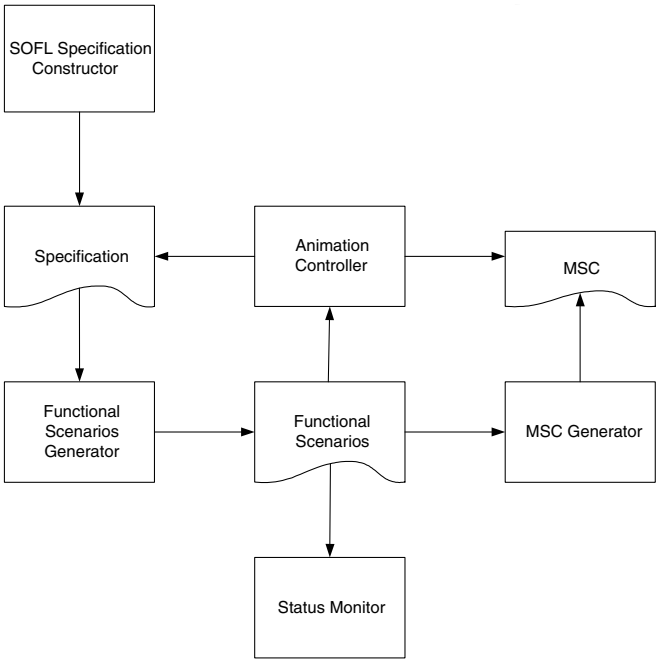


Fig. 4. The overall structure of SOFL-Animator.

4.2.2. Scenarios generation

As briefly mentioned in Section 4.1, an algorithm for generating all the possible scenarios from a CDFD has been described in our previous publication. Basically, our tool implements the algorithm, but with necessary extension to generate the related pre- and postconditions for each process in a scenario. Since the algorithm for scenarios generation is not a new contribution, we omit the detailed introduction to the algorithm in this paper. The reader who is interested in the algorithm can refer to our paper (Liu et al., 1998d). Instead, we focus on the algorithm of deriving the related pre- and postconditions of a process.

Consider the process *Check_Password* in scenario *2* in Section 3.1 as an example. Since the output data flow of scenario *2* is *cash*, only the subexpression defining the first case in the postcondition of process *Check_Password* is useful for the semantic level analysis of the scenario. We call such a sub-postcondition a *related-action postcondition* of the process with respect to the scenario. In order to enhance the efficiency of the semantic level analysis algorithm adopted in our animation tool, when the scenario is generated, actually the corresponding related-action postcondition is also obtained. A way to derive all the related-action postconditions of process A is to convert the postcondition into a disjunctive normal form (DNF) and treat each disjunctive clause as a related-action postcondition. In the case of the process *Check_Password*, its postcondition can be converted into the following DNF:

$$
\begin{aligned}
&(\exists!_{x\in Account\_file} \cdot x.id = card\_id \wedge x.password = pass) \wedge \\
&sel = false \wedge account1 = x \\
&\vee \\
&(\exists!_{x\in Account\_file} \cdot x.id = card\_id \wedge x.password = pass) \wedge \\
&sel = false \wedge account2 = x \\
&\vee \\
&(\neg\exists!_{x\in Account\_file} \cdot x.id = card\_id \wedge x.password = pass) \wedge \\
&e\_msg = \text{"}Reenter\ your\ password\ or\ insert\ the\ correct\ card\text{"}
\end{aligned}
$$

We implemented the algorithm for converting a predicate expression into an equivalent DNF described in Dick and Faivre (1993) in our tool, with the extension to deal with the compound expressions, such as **if-then-else**, **case**, and **let** expressions, on the basis of the translation rules shown in Table 1.

Furthermore, in our extended algorithm for generating scenarios from a specification we also need to deal with the structures that can be used in CDFDs, such as binary condition, multiple condition, nondeterministic, and broadcasting structures, as shown in Table 2. Since each structure models a transformation from an input data flow to output data flows, we treat a structure as a process, and

generate its pre- and postconditions based on the syntax and semantics of the structure, as defined in Table 2.

The use of the tool is quite simple. When the " Path-Search" button is clicked, the tool will automatically derive all the possible scenarios for the CDFD drawn in the CDFD window, as shown in Fig. 5. As the result, the derived scenarios are listed in the top left panel of the window. The default names for the scenarios are in the form "Path*n*" where *n* is zero or a natural number. If a scenario in the list is selected, its entities in the corresponding CDFD will be highlighted to draw the user's attention.

4.3. MSC generator

The MSC generator constructs a MSC for a selected scenario, as shown in Fig. 6. Each process is represented by a rectangle (in red color, although it may not be seen if the paper is not printed out by a color printer) and has a related-action pre- and postcondition. By activating the process property dialogue, the user can see the pre- and postconditions of the process.

To automatically draw a MSC for a given scenario, we build a set of rules for mapping the entities of a scenario in SOFL to the entities in a corresponding MSC introduced in Haugen (2001). The rules are summarized as follows:

Mapping rules:

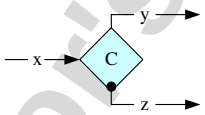
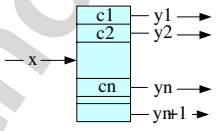
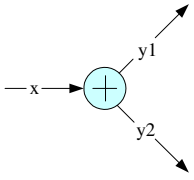
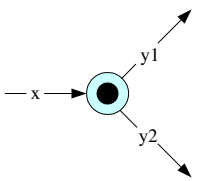
- A process in the scenario is mapped to a message-passing instance in the MSC.
- A data flow in the scenario is mapped to a message in the MSC.
- The related-action pre- and postconditions of a process in the scenario is translated into the corresponding conditions in the MSC.
- A data flow loop in the scenario is mapped to a loop box in the MSC.
- A parallel structure in the scenario is mapped to a parallel in-line box in the MSC.

Applying these rules to the following scenario described in Section 3.1: ({*balance_comm*}, {~*Account_file*}) [*Receive_Request*, *Check_Password*, *Show_Balance*]({*balance*}, {*Account_file*}) we construct the MSC in Fig. 7. In this MSC, each of the processes *Receive_Command*, *Check_Password*, and *Show_Balance* is treated as an instance; each of the data flows *balance_comm*, *sel*, *card_id*, *pass*, *account*2, and *balance* is mapped to the corresponding message in the MSC.

Table 1
The rules for translating compound expressions into DNFs

Compound expression	DNF
if c **then** o_1 **else** o_2	$(c \wedge o_1) \vee (\neg c \wedge o_2)$
case c **of** $\langle 1\rangle \rightarrow o_1$;$\langle 2\rangle \rightarrow o_2$;...;$\langle n\rangle \rightarrow o_n$	$(c1 = \langle 1\rangle \wedge o_1) \vee$ $(c = \langle 2\rangle \wedge o_2) \vee \ldots \vee$ $(c = \langle n\rangle \wedge o_n)$
let $v_1 = E_1$, $v_2 = E_2, \ldots, v_n = E_n$ **in** $P(v_1, v_2, \ldots, v_n)$	$P[E_1/v_1, E_2/v_2, \ldots E_n/v_n]$

Table 2
The major structures in a CDFD

Structure	Representation	Pre	Post
Binary condition		*true*	$C(x) \wedge y = x \vee$ $(\neg C(x) \wedge z = x)$
Multiple condition		*true*	$C1(x) \wedge y1 = x \vee$ $C2(x) \wedge y2 = x \vee \cdots \vee$ $Cn(x) \wedge yn = x \vee$ $\neg(C1(x) \vee C2(x) \vee \cdots \vee Cn(x)) \wedge yn + 1 = x$
Non-deterministic		*true*	$y1 = x \vee y2 = x$
Broadcasting		*true*	$y1 = x \wedge y2 = x$

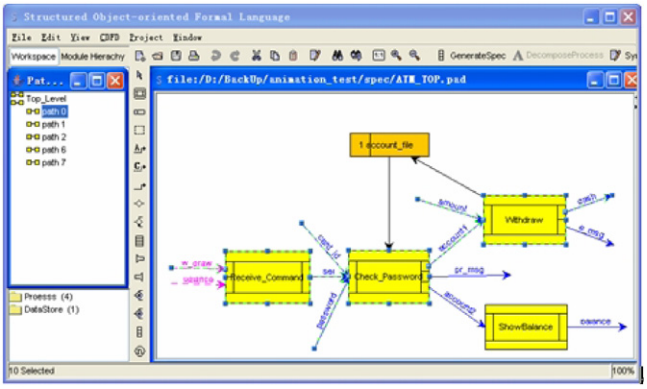


Fig. 5. A snapshot of the tool for generating functional scenarios.

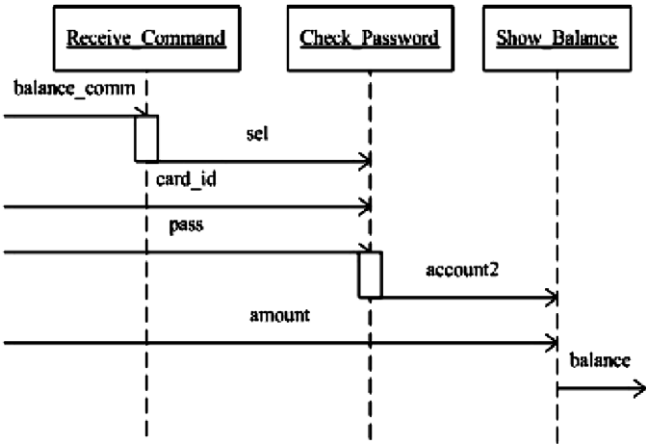


Fig. 7. A MSC for the functional scenario of showing the account balance in the ATM.

4.4. Operations of MSC

The tool supports the user to perform syntactic level analysis of a scenario in its MSC format. Since the analysis must be performed by humans, the tool is designed to provide useful operations to facilitate the user to examine every part of the animation whenever necessary. Specifically, the tool offers the monitoring operations, including "start", "pause", "forward", and "stop"; each of the operation can be executed by clicking on the corresponding button, as indicated in the snapshot of the tool in Fig. 8. The operation "start" activates the automatic animation that will not terminate until either it is interrupted by another operation or it finishes. The "pause" holds the animation, while the "forward" allows the animation to go one step further each time. The "stop" terminates the animation of the current scenario. Controlling these operations is actually similar to using a "remote controller" of a TV set to select programs of interest.

4.5. Executing a scenario

Executing a scenario is aimed at supporting the semantic level analysis and presenting a dynamic demonstration of the potential behavior of the scenario. Since all the processes in the scenario are defined using pre- and postconditions and not all such expressions can be automatically transformed into an executable program (Hayes and Jones, 1989), our animation tool requires that all the process specifications be executable. Specifically, this implies that every output variable of a process must be defined independently of other output variable definitions. For example, suppose process A is defined as follows:

process $A(x_1 : int)\ y_1 : int, y_2 : int$
pre $x_1 > 0$
post $y_1 = E_1(x_1) \wedge y_2 = E_2(x_1)$
end_process

where $E_1(x_1)$ is an integer expression containing the input variable x_1 but not the output variable y_2, and $E_2(x_1)$ is the same kind of expression, but does not contain the output variable y_1. Thus, we can easily translate the postcondition into a sequential statement in a programming language.

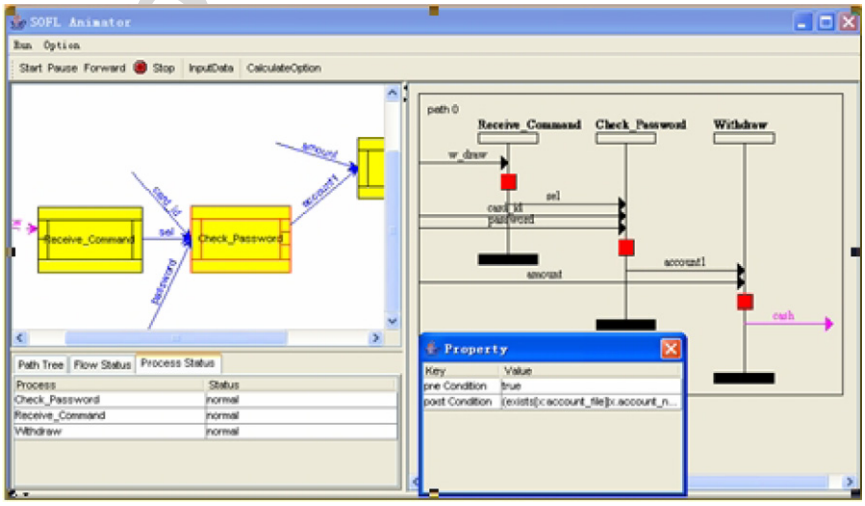


Fig. 6. A snapshot of the tool for generating a MSC.

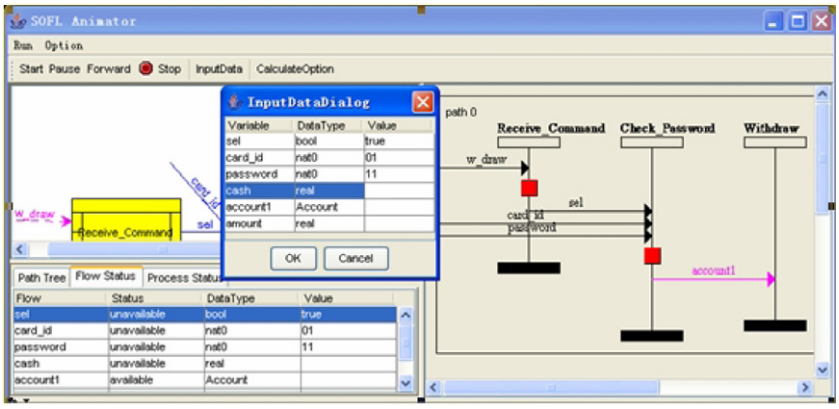


Fig. 8. A Snapshot of the tool for executing a scenario.

Our tool supports the execution of a scenario in two steps. The first step is to automatically translate the specifications of all the processes in the scenario into Java program segments. The second step is to provide assistance to the reviewer to generate test cases, to execute the scenario, and to detect errors by analyzing the execution result. Fig. 8 shows a snapshot of the tool to support the execution of a scenario.

Translation of a process specification into a Java program needs to map data types used in SOFL to types used in Java and to construct an algorithm based on the pre- and postconditions of the process. Table 3 shows examples of mapping SOFL data types into Java types, and other data type mappings are omitted for brevity.

For the construction of an algorithm to implement a process specification, we adopt the transformation approach described in our previous publication (Liu and Ho-Stuart, 1996). The essential idea of the approach is to first convert the postcondition of the process into an equivalent disjunctive normal form (e.g., $Q_1 \vee Q_2 \vee \cdots \vee Q_n$) and then translate the disjunctive normal form into a conditional statement. In such a disjunctive normal form, each disjunctive clause is a conjunction of a "guard condition" and a "defining condition" (e.g., $Q_1 = G_1 \wedge D_1$, where G_1 is a guard condition and D_1 is a defining condition). A guard condition is a predicate that contains no output variable of the process, while a defining condition is also a predicate that must contain only one output variable (of course, it may also contain input variables) of the process.

Table 3
The examples of data type mapping

Type in SOFL	Type in Java
bool	boolean
nat, nat0, int	int
real	float
Set	HashSet
Map	HashMap
char	char
string	String
Sequence	ArrayList

This means that the defining condition shows how the output variable is defined based on input variables of the process. Since a systematic description of the transformation approach, including algorithms and rules, is available in our previous publication (Liu and Ho-Stuart, 1996), we do not repeat the details of the approach here for brevity.

5. A small case study

In order to assess the performance and the usability of our animation tool, we have conducted a small case study animating the specification of a simplified university library system. The system is required to provide the most important and basic services a normal library usually offers, such as "students registration", "book registration and management", "borrowing books", "returning books", "renewing books", "searching books", and "fine for overdue". The specification of the system was written in the SOFL specification language by a group of graduate students, and it was informally reviewed by an independent reviewer for its validity after it was completed about one year ago from the time when the case study was conducted.

Our case study is aimed at investigating how effectively the tool can help the animation process in a user-friendly manner. Instead of pursuing a rigorous assessment through a well-defined scientific experiment, we expected to gain the knowledge and evidence of the performance of the tool by simply using the tool. As the result, we detected three errors in relation to process decomposition, 7 errors concerned with consistency between CDFDs and their module specifications, and 12 errors in using variables, as shown in Table 4. An error in process decomposition means that a scenario in a process specification is not correctly implemented by the decomposed CDFD of the process. An error violating the consistency between a CDFD and its module specification is a defect that leads to the failure of finding the related-action pre- and postconditions of some processes for a scenario. An error in variables means that some variable used in the specification is not appropriate (e.g., the same variable is used as two input data flow variables for a process).

Our experience of using the tool to detect 22 errors in the specification, which were not found by the informal review before, suggests that the animation tool is effective in helping the user reveal high level structural errors. The reason for the high effectiveness is that using our animation tool enables the user to analyze every possible scenario and to inspect every process and the related data items in the scenario. Furthermore, we have also found that the tool is easy to use due to the navigation among scenarios, status table, scenario and status highlighting function, and the MSC representation.

On the other hand, the tool has limitation in supporting the semantic level analysis. Since not all the compound data types provided in the SOFL specification language, such as union types and composite types, are implemented in the tool, executing a scenario may not be performed automatically if some processes in the target scenario are specified using those types. In that case, only the syntactic level analysis can be conducted. Another limitation is that the tool cannot generate properly all the possible scenarios for CDFDs with many nested data flow loops. However, since the number of data flow loops in a CDFD can be reduced by introducing more abstraction in the module specification of the CDFD according to our experience of modeling using SOFL before (Liu et al., 1998a; Liu et al., 1999), the later limitation can be controlled to some extent. Although these limitations are currently with the tool, there is no reason why we cannot resolve them with further efforts in the future.

It would be more valuable if we could carry out an experiment to compare our animation approach to some of the existing animation techniques, such as Miller and Strooper's approach (Strooper and Miller, 2001). However, performing such a comparison for a reasonable conclusion requires complicated and quality activities, such as correctly translating the specification in SOFL into the one in the language which is supported by the corresponding tool, appropriately selecting animation cases, and accurately analyzing the animation results obtained by using the different approaches and tools. It seems extremely difficult to reach a fair conclusion based only on a small-scale experiment with many uncertain elements involved (e.g., how satisfactory is the translation of the specification? how familiar are the reviewers using the different animation approaches with the domain and the specification of the system?). We believe that a systematic, rigorous, and large-scale experiment for a fair comparison between our approach and the existing ones is an important work and requires relatively long-time efforts, which is beyond the scope of this paper. We plan this work as one of our future research projects.

Table 4
Table caption

Error Category	Number of errors detected
Errors in process decomposition	3
Errors violating the consistency between CDFD and module specification	7
Errors in variables	12

6. Related work

Animation of formal specifications has drawn great attention from research community because it provides a considerable potential to help human understand the behaviors of the systems defined in their specifications and analyze whether the specifications accurately capture the intended requirements. In this section, we introduce related work on specification animation and discuss its similarity to and difference from our animation approach described in this paper.

The most commonly used approach to specification animation is first to transform an executable subset of the target formal specifications to be animated into a program either in a logic or functional language, and then execute specifications (indirectly) with test cases or animation cases (values for input variables). Morrey and his colleagues built a toolset to support the construction and animation of Z formal specifications (Morrey et al., 1998). The toolset is composed of two subtools, known as *wiZe* and *ZAL*, respectively. *wiZe* supports the construction of *Z* specifications (including editor, syntax and type analyzer) and transforms them into an executable representation in an extended LISP, while ZAL allows the user to animate the specifications by executing the corresponding program in LISP. Diller describes a systematic approach to animating *Z* specifications in Diller (1994). By this approach a *Z* specification is first translated into a functional program in Miranda, and then the specification is executed (indirectly) for animation purpose. Since the first-order logic adopted by *Z* is undecidable, *Z* specifications are not necessarily executable in general. For this reason, the animation approach is only applicable to an executable subset of *Z* specifications. Other similar approaches to the animation of *Z* specifications are presented in Abdallah et al. (2003), Sherrell and Carver (1994), Hewitt et al. (1997). Slightly differing from the above work, Kazmierczak and his colleagues describe another approach to verifying *Z* specifications that combine static analysis and animation to systematically explore properties of the specifications (Kazmierczak et al., 1998). Specifically, the approach first translates *Z* schemas to predicate expressions and then conduct a static analysis to identify the executable expressions. The final step is to "execute" the identified predicate expressions by substituting concrete values for corresponding variables in the predicate expressions and evaluate the expressions. In addition to the above work on *Z* specification animation, there are also studies on animation of other specification languages, such as OCL for UML (Gray and Schach, 2000), B (Waeselynck and Behnia, 1998), and SUM (Hazel et al., 1997).

Compared to the above "transformation and testing" approach, another kind of animation approach is to adopt

MSC as a framework to provide a graphical user interface to represent animation. Such an approach allows the user of animation tools to monitor the process of animation by analyzing interactions between operations or instances through exchange of messages. Combes and his colleagues describe an open animation tool for telecommunication systems (Combes et al., 2002). The tool is known as ANGOR, and it offers an environment based on a flexible architecture. It allows animating different animation sources, such as formal and executable language like SDL and scenario languages like MSC. Stepien and Logrippo constructed a toolset for animation of LOTOS execution traces (Stepien and Logrippo, 2002). The toolset includes a translator from LOTOS traces to MSC and a graphic animator. The translator was implemented by providing certain mappings between the elements of LOTOS actions and the elements of the MSC that provides a visual overview of all message exchanges. The graphic animator works on the generated MSC by providing only one message exchange at a time.

Some researchers exploited the way of utilizing model checker or static analyzer to facilitate animation of specifications. Gargantini and Riccobene proposed an automatic model driven approach to animating formal specifications in Parnas' SCR tabular notation (Gargantini and Riccobene, 2003). By this approach necessary scenarios, called animation sequences, defined in requirements specifications are first derived and then those scenarios are used to animate critical system behaviors through a graphical interface. 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. Jackson and his colleagues describe a tool called *Alcoa* that supports the construction of formal specifications in Alloy and checks the consistency of the specifications by analyzing whether the assertions of the specifications are satisfied at every step of specification animation (Jackson et al., 2000).

Compared to the existing work on specification animation above, our approach features the combination of static analysis (i.e., syntactic level analysis) and dynamic analysis (i.e., semantic level analysis). This approach enables the user of the animation tool to review how data items flow among processes in the specification and to analyze whether the involved data items and processes are correctly designed in the specification. It can also allow the user to monitor the dynamic execution of functional scenarios derived from the specification at every process step (that is, before and after the execution of the process), thus the user will have an opportunity to examine the semantics of the processes. Through the activities of monitoring and analysis during the animation, the user is expected to understand the specification and to detect errors contained in the specification. Furthermore, our tool is one of very few existing tools for animation of data flow formal specifications, and the first one for the SOFL specification language. We believe that our work described in this paper will contribute to the maturity of the SOFL method and to the application of animation to data flow specifications in other languages (e.g., FOCUS (Broy and Stolen, 2001)).

7. Conclusion and future work

We have proposed an approach to animating a formal specification by animating every possible functional scenario defined in the specification. Animation of a scenario can be conducted at both syntactic and semantic levels to help the reviewer understand the potential behaviors of the specification and to detect errors in relation to the consistency and validity of the specification. We have also developed a software tool to support the animation approach. The tool offers functions for the entire animation process, including generating functional scenarios from a CDFD, controlling the syntactic and sematic level analysis, and managing animation data. We have used the tool to animate a simplified library system specification written in the SOFL specification language. The result shows that the tool is effective in assisting the user to understand the specification and to detect errors, although some limitations of the tool exist.

We will continue to improve the current animation technique and tool in our future research. It includes resolving the limitations of the current tool in dealing with compound data types and nested data flow loops for generating functional scenarios, and conducing a systematic and large-scale experiment for a rigorous comparison between our animation approach and the existing ones. We are also interested in developing a new technology by integrating the animation approach with specification review (Liu, 2003c,d) and specification testing (Liu, 1999). We believe that the three different approaches are complementary and an appropriate integration of them will lead to a more cost-effective and practical technology for the verification and validation of formal specifications and designs of software systems.

Acknowledgments

This work is supported in part by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research on Priority Areas (No. 16016279), Grant-in-Aid for Scientific Research (No. 18500027), and Software School at Shanghai Jiaotong University.

References

Abdallah, A.E., Bowen, J.P., Barros, A., Barros, J.B., 2003. A provably correct functional programming approach to the prototyping of formal Z specifications. Proceedings of ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'03), Tunis, Tunisia, July 14–18. IEEE Computer Society Press.

Boehm, B.W., 1981. Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ.

Bowen, J., Hinchey, M.G., 1999. High-Integrity System Specification and Design. In: FACIT Series. Springer, Berlin.

Broy, Manyfred, Stolen, Ketil, 2001. Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer, Berlin.

Chen, H., Shen, Y., Jiang, J., 2005. Extended SOFL features for the modeling of middleware-based transaction management. Proceedings of 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS2005), Presented at the Workshop on SOFL, Shanghai, China. IEEE Computer Society Press, pp. 16–17.

Combes, P., Dubois, F., Renard, B., 2002. An open animation tool: application to telecommunication systems. Computer Networks: The International Journal of Computer and Telecommunications Networking 40 (5), 599–620.

Dick, J., Faivre, A., 1993. Automating the generation and sequencing of test cases from model-based specifications. Proceedings of FME'93: Industrial-Strength Formal Methods, Odense, Denmark. In: Lecture Notes in Computer Science, vol. 670. Springer, Berlin, pp. 268–284.

Diller, A., 1994. Animation Using Miranda. Wiley, New York, pp. 271–278 (Chapter 19).

Gargantini, A., Riccobene, E., 2003. Automatic model driven animation of SCR specifications. In: Mauro Pezzè (Ed.), Fundamental Approaches to Software Engineering, 6th International Conference (FASE 2003), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2003), Warsaw, Poland, April 7–11, . In: Lecture Notes in Computer Science, vol. 2621. Springer, Berlin.

Gomaa, H., 2000. Designing Concurrent, Distributed, and Real-Time Applications with UML. Pearson Education, Inc.

Gomaa, H., Liu, S., Shin, M., 2000. Integration of domain modeling method for families of systems with the SOFL formal specification language. Proceedings of Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS2000), Tokyo, Japan, September 11–14. IEEE Computer Society Press.

Gray, J., Schach, S., 2000. Constraint animation using an object-oriented declarative language. Proceedings of the 38th Annual ACM Software Engineering Conference, Clemson, SC, US, April 7–8. ACM Press, pp. 1–10.

Haugen, O., 2001. MSC-2000 interaction diagrams for the new millennium. Computer Networks (35), 721–732.

Hayes, I., Jones, C.B., 1989. Specifications are not (necessarily) executable. Software Engineering Journal 4 (6), 330–339.

Hazel, D., Strooper, P.A., Traynor, O., 1997. Possum: an animator for the SUM specification language. Proceedings of 1997 Asia-Pacific Software Engineering Conference (APSEC'97). IEEE CS Press, pp. 42–51.

Hewitt, M., O'Halloran, C., Sennett, C., 1997. Experiences with PiZA: an Animator for Z. Proceedings of 1997 Z User Meeting (ZUM'97). In: Lecture Notes in Computer Science, vol. 1212. Springer, Berlin, pp. 37–51.

Hinchey, M.G., Bowen, J. (Eds.), 1999. Industrial-Strength Formal Methods in Practice. FACIT Series. Springer, Berlin.

Jackson, D., Schechter, I., Shlyakhter, I., 2000. Alcoa: the alloy constraint analyzer. In: Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, pp. 730–733.

Jones, C.B., 1990. Systematic Software Development Using VDM, second ed. Prentice-Hall, Englewood Cliffs, NJ.

Kazmierczak, E., Winikoff, M., Dart, P., 1998. Verifying model oriented specifications through animation. Proceedings of 1998 Asia Pacific Software Engineering Conference. IEEE Computer Society Press, pp. 254–261.

Leveson, N.G., 2000. Intent specifications: an approach to building human-centered specifications. IEEE Transactions on Software Engineering 26 (1), 15–35.

Linger, R.C., Trammell, C.J., 1999. Cleanroom software engineering: theory and practice. In: Hinchey, M.G., Bowen, J.P. (Eds.), Industrial-Strength Formal Methods in Practice. Springer, Berlin, pp. 351–372.

Liu, S., 1999. Verifying consistency and validity of formal specifications by testing. In: Wing, J.M., Woodcock, J., Davies, J. (Eds.), Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, . In: Lecture Notes in Computer Science. Springer, Berlin, pp. 896–914.

Liu, S., 2003a. Formal verification of condition data flow diagrams for assurance of correctness network protocols. Proceedings of 17th International Conference on Advanced Information Networking and Applications (AINA'03), Xi'an, China. IEEE Computer Society Press, pp. 289–292.

Liu, S., 2003b. A case study of modeling an ATM using SOFL. Technical Report HCIS-2003-01, CIS, Hosei University, Koganei-shi, Tokyo, Japan, 2003.

Liu, S., 2003c. A property-based approach to reviewing formal specifications for consistency. In: Proceedings of 16th International Conference on Software and Systems Engineering and their Applications, Paris, France, December 2–4, vol. 4, pp. 1/6–6/6.

Liu, S., 2003d. A rigorous approach to reviewing formal specifications. Proceedings of 27th Annual IEEE/NASA International Software Engineering Workshop, Greenbelt, USA, December 4–6. IEEE Computer Society Press, pp. 75–81.

Liu, S., 2004a. A survey on the use of SOFL based on four projects. Technical Report HCIS-2004-01, CIS, Hosei University, Koganei-shi, Tokyo, Japan, 2004.

Liu, S., 2004b. Formal Engineering for Industrial Software Development Using the SOFL Method. Springer, Berlin, ISBN 3-540-20602-7.

Liu, S., Ho-Stuart, C., 1996. Semi-automatic transformation from formal specifications to programs. Proceedings of the International Conference on Engineering of Complex Computer Systems, Montreal, Quebec, Canada, October 21–25. IEEE Computer Society Press, pp. 506–513.

Liu, S., Asuka, M., Komaya, K., Nakamura, Y., 1998a. An approach to specifying and verifying safety-critical systems with practical formal method SOFL. Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'98), Monterey, CA, USA, August 10–14. IEEE Computer Society Press, pp. 100–114.

Liu, S., Asuka, M., Komaya, K., Nakamura, Y., 1998b. Applying SOFL to specify a railway crossing controller for industry. Proceedings of 1998 IEEE Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98), Boca Raton, FL, USA. IEEE Computer Society Press.

Liu, S., Offutt, A.J., Ho-Stuart, C., Sun, Y., Ohba, M., 1998c. SOFL: a formal engineering methodology for industrial applications. IEEE Transactions on Software Engineering 24 (1), 337–344, Special Issue on Formal Methods.

Liu, S., Offutt, J.A., Ohba, M., Araki, K., 1998d. The SOFL approach: an improved principle for requirements analysis. Transactions of Information Processing Society of Japan 39 (6), 1973–1989.

Liu, S., Shibata, M., Sat, R., 1999. Applying SOFL to develop a university information system. Proceedings of 1999 Asia-Pacific Software Engineering Conference (APSEC'99), Takamatsu, Japan, December 6–10. IEEE Computer Society Press, pp. 404–411.

Mills, H.D., Dyer, M., Linger, R.C., 1987. Cleanroom software engineering. IEEE Software 5 (4), 19–25.

Morrey, I., Siddiqi, J., Hibberd, R., Buckberry, G., 1998. A toolset to support the construction and animation of formal specifications. Journal of Systems and Software 41 (3), 147–160.

Offutt, A.J., Liu, S., 1999. Generating test data from SOFL specifications. Journal of Systems and Software 49 (1), 49–62.

Shen, Y., Chen, H., 2005. Extending SOFL features for AOP modeling. Proceedings of 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS2005), Presented at the Workshop on SOFL, Shanghai, China. IEEE Computer Society Press, pp. 14–15.

Sherrell, L.B., Carver, D.L., 1994. Experiences in translating Z designs to Haskell implementations. Software Practice and Experience 24 (12), 1159–1178.

Stepien, B., Logrippo, L., 2002. Graphic visualization and animation of LOTOS execution traces. Computer Networks: The International Journal of Computer and Telecommunications Networking 40 (5), 665–681.

Strooper, P.A., Miller, T., 2001. Animation can show only the presence of errors, never their absence. Proceedings of 2001 Australian Software Engineering Conference. IEEE CS Press, pp. 76–88.

Taguchi, Y., 2000. Formal specification of network protocol using the SOFL specification language. ITPC Project Report, ITPC, Faculty of Engineering, Hosei University, Tokyo, Japan, 2000.

Waeselynck, H., Behnia, S., 1998. B model animation for external verification. Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98). IEEE CS Press, pp. 36–45.

Warmer, Jos, Kleppe, Anneke, 1999. The Object Constraint Language: Precise Modelling with UML. In: Object Technology Series. Addison-Wesley.

Xue, X., 2005. A formal specification constructing tool for SOFL. Proceedings of 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS2005), Presented at the Workshop on SOFL, Shanghai, China. IEEE Computer Society Press, pp. 12–13.

Yourdon, E., 1989. Modern Structured Analysis. Prentice-Hall, Englewood Cliffs, NJ.