The Journal of Systems & Software 178 (2021) 110948

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A formal specification animation method for operation validation ☆,☆☆

Shaoying Liu [a], Weikai Miao [b,*]

[a] Graduate School of Advanced Science and Engineering, School of Informatics and Data Science, Hiroshima University, Japan
[b] Software Engineering Institute, East China Normal University, China

ARTICLE INFO

Article history:
Received 29 July 2020
Received in revised form 1 December 2020
Accepted 8 March 2021
Available online 15 April 2021

Keywords:
Formal specification
Specification animation
Verification and validation

ABSTRACT

Formal specification can benefit software quality by precisely defining the behaviors of operations to prevent primary mistakes in the early phase of software projects, but a remaining challenge is how such a specification can be checked comprehensibly to show whether it satisfies the user's perception of requirements. In this paper, we describe a new technique for animating operation specifications as a means to address this problem. The technique offers new ways to do (1) automatic animation data generation for both input and output of an operation based on pre- and post-conditions, (2) visualized demonstration of the relationships between input and the corresponding output, (3) comprehensible animation of data items, and (4) illustrative animation of logical expressions and the operators used in them. We discuss these issues and present a prototype tool that supports the automation of the proposed technique. We also report an industrial application as a trial experiment to validate the technique. Finally, we conclude the paper and point out future research directions.

1. Introduction

Formal specification is a technique for defining what to be done by a potential software system using a mathematically-based language or notation. If used properly to appropriate application domains, formal specification can produce benefits in terms of improving software reliability and reducing the cost of its development (Kurita et al., 2008a; Woodcock et al., 2009). However, formal specification also faces challenges that keep many practitioners away from utilizing its advantages (Knight et al., 1997; Parnas, 2010). Our experience in applying formal specification in industry (Liu et al., 1998a; Luo et al., 2016) has made us realize that formal operation specifications are the places where mistakes often occur.

In realistic software projects, formal specifications are mainly written and read by the developers who are well trained in formal methods, but they may occasionally be used as a vehicle for communication between the developer and some users who have a good background of mathematics or formal methods. However, as far as our experience in Japan and China can tell, most users of software systems developed in industry would not be expected to understand formal specifications (Liu et al., 2008). Therefore, the formal specification written by the developer will need to be validated against the user's requirements in a comprehensible manner.

Specification animation was proposed as a means for formal specification validation in 1990's (Hazel et al., 1997; Miller and Strooper, 2003b). The essential idea is trying to make formal specifications in pre- and post-conditions executable so that sample input data, which is called *animation data*, can be properly selected to demonstrate the behavior dynamically. But since the proposed approach requires automatic translation from formal specifications in a formal notation (e.g., Z) to code, it can only deal with a subset of the formal notation because not all of the specifications in pre- and post-conditions can be automatically refined into code for execution (Hayes and Jones, 1989). Almost in the same period, Chen and Liu proposed an alternative technique called *specification testing* that does not require translation of formal specifications to code but can automatically generate test data for input variables and expected output data from pre- and post-conditions (Chen and Liu, 1996; Liu, 1999a). Although this work has set up an important foundation for building a more useful technique for specification animation, no much progress in technical details was made until we undertake the study presented in this paper. The main difference between the specification testing proposed previously and the specification animation proposed in this work is that the latter supports visualized demonstration of both operation behaviors and data characteristics for validation while the former only focuses on the generation of input and output values for an operation to check the consistency of its specification.

As is well known, the animation of software specifications for validation is notoriously difficult, primarily because it requires

☆ This work was supported by JSPS, Japan KAKENHI Grant Number 26240008 and NSFC of China 61402178 and 61872144.
☆☆ Editor: David Shepherd.
* Corresponding author.
E-mail address: wkmiao@sei.ecnu.edu.cn (W. Miao).

https://doi.org/10.1016/j.jss.2021.110948
0164-1212/

that humans understand what the animation is demonstrating. Furthermore, animation also faces another technical difficulty due to the fact that logical formulas and expressions often contain nonconstructive and abstract elements. To deal with these challenges, the strategy in our proposed animation technique is to generate comprehensible and representative animation data based on the pre- and post-conditions of an operation to visually demonstrate the relationships between input and output values and the characteristics of the values for the user who is usually interested in the behavior of the system under construction. For the analyst who writes the specification, we choose to animate the process of evaluating designated logical expressions to help him or her confirm whether the expressions are specified as desired.

This paper discusses the specification animation technique in four aspects: (1) how animation data for both input and output of an operation can be automatically generated based on its pre- and post-conditions, (2) how a visualized demonstration of the operation behaviors in terms of the input–output relation can be conducted, (3) how input and output variables of the operation can be confirmed to be properly declared with appropriate types, and (4) how the logical expressions involved in the pre- and post-conditions of the operation can be ensured to be valid.

We have made the following three contributions in this paper:

- A novel automatic animation data generation technique is proposed.
- A tool-supported approach to animating operations, data items, and logical expressions is established.
- An industrial application is conducted to evaluate the proposed animation technique in terms of detecting faults in specifications.

In general, a software system is composed of more than one operation, therefore mistakes may occur in the integration of operations. For this reason, both the integration of operations and each individual operation in formal specifications need to be validated. We have put forward an approach to handling the animation of operation integrations in formal specifications in a previous work (Li and Liu, 2015). On the other hand, in many cases of realistic software engineering projects, a specification is often written as a set of rather independent operations and each operation is specified separately. Integrating the operations into an architecture is often regarded as part of software design rather than specification. This style becomes more and more adopted in the Agile development paradigm where comprehensive documentation is not emphasized. Of course, different schools may have different opinions on this since the Agile paradigm has both advantages and disadvantages. Our position is on the side of the combination of formal specification with the Agile principles under necessary compromises from both sides because our work on the Agile formal engineering methods (Liu, 2018) has shown that such a combination helps enhance both software productivity and quality. In this paper, we only focus on the discussion of the animation of individual operations specified in pre- and post-conditions. As mentioned above, our experience suggests that faults should often be introduced into individual operation specifications due to the difficulty in achieving abstract but precise mathematical expressions with high complexity.

The remainder of the paper is organized as follows. Section 2 briefly describes the goal of specification animation. Section 3 discusses the issues in relation to animation data generation. Section 4 presents a series of algorithms for generating animation data from various logical expressions in formal specifications. Section 5 focuses on a supporting tool we have built for our animation approach. Section 6 presents an industrial application of our technique to confirm its feasibility and effectiveness. Section 7 reviews related work and compare our work with existing work. Finally, in Section 8, we conclude the paper and discuss the future research topics.

2. Goal of specification animation

Given an operation specification in pre- and post-conditions, the goal of the specification animation is to analyze and confirm the functionality of the operation defined in the specification, the characteristics of the related data items, and the meaning of the formal expressions involved in the formal specification through visualized demonstrations.

2.1. Key issues

Since the determination of whether a specification meets the user's requirements usually needs human judgment, one of the key issues to be addressed is what aspects of the specification needs to be visually demonstrated so that the user would feel easy to make correct judgments. On the basis of our long-term experience in collaborating with industry, we have realized that visualized demonstration of the input–output relation, the characteristics of the data items involved, and the meaning of the formal expressions in the specification can significantly help the user comprehend what the operation is intended to do and understand whether they are desirable.

Another key issue is what animation data need to be used and how they can be generated. Our essential idea is that the animation data must allow the user to check all of the important functional scenarios (similar to the concept of use cases in UML but with more precise and detailed contents) and must be generated based on the formal specification of the operation. The technical details of generating animation data will be discussed in Section 3.

The final key issue is how to comprehensibly represent the three aspects of the operation as mentioned above. Since this issue is also related to human judgment, it seems to be extremely difficult to come up with a "magic" solution. Taking the well-known view of graphical representations being more comprehensible than texts ("A picture is worth a thousand words"), we believe that visualized demonstration based on graphical representations can be helpful. The details of this point will be discussed in Section 5.

2.2. Specification language

Although the technical contributions of this paper are generally independent of specification languages in which the operation specification is written, we still need a specific language as the vehicle to represent our discussions. In this work, we choose SOFL (Structured Object-Oriented Formal Language) (Liu et al., 1998c; Liu, 2004) partly because of our expertise in SOFL and partly because a process specification in SOFL shares the same concept and style as an operation specification in the well-known formal notations (e.g., VDM, B-Method). Therefore, our discussions in this paper can also be applicable to existing similar formal notations.

A process in SOFL models a transformation from input to output (Liu, 2004). It is similar to an operation in VDM but has a more general structure to allow multiple input and output ports. This structure enables a process to be used flexibly for abstraction in modeling systems. This point will become clearer as the discussion on the structure of a process and its specification progresses.

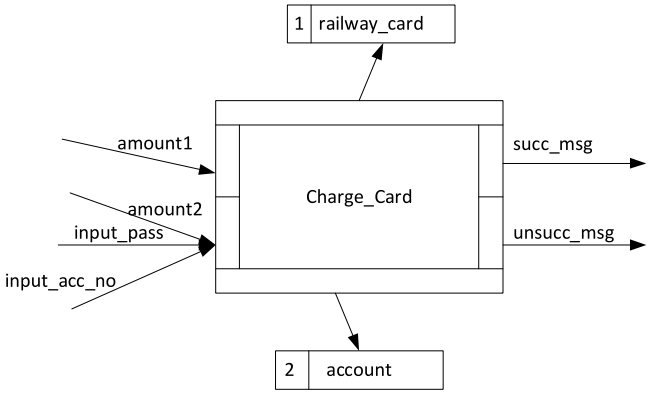

Fig. 1. A process for charging railway card.

A process is represented in both graphical and textual formats that are complementary for understanding. For instance, Fig. 1 shows a graphical representation of a process named *Charge_Card* in a railway card system we have specified in our previous project (Liu, 1999b). For the purpose of explaining the basic concepts involved in our discussions, we have simplified the original version of the process specification to the extent that we believe ensures a good understandability.

The process has two input ports represented by the two small narrow rectangles on the left side of the graphical representation, and two output ports represented by the two small narrow rectangles on the right side. The upper input port receives one input data flow *amount*1, representing the amount of input cash, while the lower input port receives three input data flows *amount*2 (the amount to be charged to the *railway_card* from the customer's bank account), *input_pass* (the password supplied by the customer for accessing to his or her bank account), and *input_acc_no* (the account number provided by the customer).

One important feature of the process is that all of the input ports (or all of the output ports) are *exclusive* in terms of using their input data flows (or producing their output data flows). There are some rules defined in the SOFL language for writing pre- and post-conditions of a process with multiple input/output ports. These rules disallow the data flow variables of different ports to join the same conjunction. Such a rule is enforced through a consistency checking supported by a tool developed in our another project before (Liu et al., 2010). For example, the process *Charge_Card* is intended to charge the railway card either by cash or through the customer's bank account, but not both. The railway card is denoted by the data store named *railway_card* and the customer's bank account is represented by another data store named *account*. Each store is assigned a number such as 1 or 2 for reference purpose in communication between the developers. The process is connected to the two stores in a way that they can be updated by the process. If the *amount*1 becomes available (i.e., is supplied), the process takes it as input and adds it to the balance of the *railway_card*. If the *amount*1 is not available but the *amount*2, *input_pass*, and *input_acc_no* are all available, the process accepts them as input and properly updates the *railway_card* and the *account*, according to different conditions these three arguments satisfy. The details of the conditions and the corresponding updating of the two data stores are given in the formal specification of the process as shown below.

process *Charge_Card*(
*amount*1 : *real* |
*amount*2 : *real*,
input_pass,
input_acc_no : *seq of* 0..9
)
succ_msg : *string* |
unsucc_msg : *string*
ext wr *railway_card* : *composed of*
holder_name : *string*
balance : *real*
end
wr *account* : *composed of*
acc_no : *seq of* 0..9
/*a sequence of 9 digits chosen from 0 to 9*/
password : *seq of* 0..9
/*a sequence of 6 digits chosen from 0 to 9 */
balance : *real*
end
pre *amount*1 >= 0 *or amount*2 >= 0
post *railway_card* =
modify(˜*railway_card*, *balance*− > ˜*railway_card.balance* + *amount*1) *and*
account = ˜*account and succ_msg* = "*Charge is successful*!"
or
input_pass = ˜*account.password and*
input_acc_no = ˜*account.acc_no and*
*amount*2 <= ˜*account.balance and*
railway_card =
modify(˜*railway_card*, *balance*− > ˜*railway_card.balance* + *amount*2) *and*
account = *modify*(˜*account*, *balance*− > ˜*account.balance* − *amount*2) *and*
succ_msg = "*Charge is successful*!"
or
not(*input_pass* = ˜*account.password and*
input_acc_no = ˜*account.acc_no and*
*amount*2 <= ˜*account.balance*) *and*
railway_card = ˜*railway_card and*
account = ˜*account and unsucc_msg* = "*Charge is not successful*!"
end_process;

In the signature of the process, the declarations of data flows of different input (or output) ports are separated using the vertical bar |. For instance, the declaration of the input data flow *amount*1 is separated from that of the input data flows *amount*2, *input_pass*, and *input_acc_no*. The data store variables *railway_card* and *account* are declared as writable *external variables* (similar to global variables), which is indicated by the keyword **wr** (write) after **ext** (external). Each of the two stores is defined as an object of the corresponding composite type with several fields as shown in the specification. The pre-condition of the process requires that the related input be greater than or equal to 0 when they are ready for use by the process. The post-condition defines how the *railway_card* and *account* are updated by charging the card with cash and through transfer from the bank account, respectively. In the latter case, the situation where the customer's input data are acceptable and the situation where the input data are unacceptable are specified separately. Note that a decorated store variable in the post-condition, such as ˜*railway_card*, is used to represent the initial value of the variable before the process while the undecorated store variable, *railway_card* for example, is used to denote the updated value of the variable after the process.

On the basis of this example, we give a general definition of a process next.

Definition 1. A process is a six-tuple (P, P_I, P_O, P_E, P_{pre}, P_{post}), where P is the process name, P_I a collection of the input variable sets, P_O a collection of the output variable sets, P_E a set of the external variables including both decorated and undecorated external variables (if any), P_{pre} the pre-condition, and P_{post} the post-condition of the process.

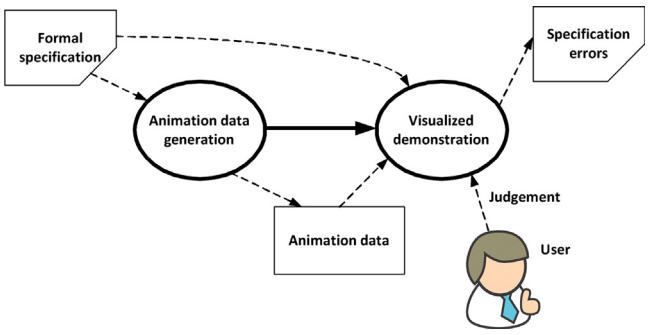

Fig. 2. Process specification animation procedure.

For instance, all of the elements of the process *Charge_Card* are as follows:

$$
\begin{aligned}
Charge\_Card_I &= \{\{amount1\}, \{amount2, input\_pass, input\_acc\_no\}\}\\
Charge\_Card_O &= \{\{succ\_msg\}, \{unsucc\_msg\}\}\\
Charge\_Card_E &= \{railway\_card, account, \tilde{}railway\_card, \tilde{}account\}\\
Charge\_Card_{pre} &= \text{the same as given above}\\
Charge\_Card_{post} &= \text{the same as given above}
\end{aligned}
$$

It is worth mentioning that a process in SOFL specifications can be decomposed into a formalized data flow diagram known as *Condition Data Flow Diagram* (CDFD) together with its associated module, in order to deal with the specification of complex and/or large scale systems (Liu, 2004). The process and its decomposition are required to be kept consistent to ensure that the behaviors defined by the decomposed CDFD and its module must satisfy the specification of the process. Such a consistency is well defined and can be verified through rigorous inspection as discussed in our previous work (Liu et al., 2010). Since our work presented in this paper focuses on the validation of a single process, we do not extend our discussion to the issue concerned with process decomposition.

2.3. Process specification animation

As mentioned previously, a process specification animation aims to check and confirm whether the specification is desirable with respect to the user's requirements and the analyst's understanding through visualized demonstrations of the input–output relation of the process, the characteristics of the related input/output data items, and the meaning of formal expressions written in the specification, using selected sample data. The procedure of carrying out an animation in our method usually takes two steps, as illustrated in Fig. 2. The first step is *animation data generation* and the second is *visualized demonstration*.

Animation data generation is aimed at selecting representative values for all of the variables involved in the specification, including input variables, output variables, and external variables (like state variables used in other formal notation). This point is similar to test data generation in program testing where both the input values and expected output values are usually prepared in advance, but the difference is that in program testing, the expected output values are not used for executing the program (although they are used for test result analysis) while the output values generated for specification animation must be used during the animation process, see more detailed discussions in Section 3.

Animation data must be generated based on the formal specification of the process. The generation from the formal specification is similar to white-box testing for programs; it must take the structure of the specification into account in order to achieve desirable functional coverage for validation. A process in the SOFL language may have multiple input ports that accept input data non-deterministically. Precisely speaking, only one input port can accept the input data for one execution of the process, but which one of the input ports will be used to execute the process is non-deterministic. This special structure may affect the evaluation of the precondition of the process and will have to be taken into account when animation data are generated, which is discussed in detail in Section 3. The activity of animation data generation may lead to the identification of faults because the process of generating animation data inevitably "force" the generator (either human or machine) to scrutinize the formal specification. That is, *animation data generation can play the role of inspecting the specification* in addition to preparing data for visualized demonstration.

Visualized demonstration for specification animation aims to demonstrate the three aspects of a process in a visualized fashion in order to facilitate both the analyst (who wrote the specification) and the user to check whether the potential behaviors of the process are desirable.

It is worth mentioning that our animation approach is intended to offer a facility for the analyst and the user to work together to identify faults and confirm the validity of the specification. Since the technique is a testing-based approach with visualized demonstration and its effect is dependent on human judgment, its application would be difficult to guarantee the correctness of the specification or any desirable efficiency in finding faults. As indicated by the three criteria for animation data generation given in the next section, our approach suggests a minimum level of animation with the selected animation data, which is to ensure at least one animation for every desired functional scenario defined in the specification. Even if this validation does not help find all of the potential faults in the specification, it can strengthen the communication between the analyst and the user. It can also enhance the confidence of the user in the validity of the specification.

3. Animation data generation from specifications

We first discuss the structure of a process specification and then proceed to describe the criteria for animation data generation based on the structure.

3.1. Functional scenarios

We assume that the pre- and post-conditions of any given process specification is in a disjunctive normal form.

Definition 2. Let $P_{pre} = P_1 \vee P_2 \vee \cdots \vee P_n$ and $P_{post} = Q_1 \vee Q_2 \vee \cdots \vee Q_m$ be both a disjunctive normal form. Then, we call a conjunction $P_i \wedge Q_j$ $(i = 1, \ldots, n; j = 1, \ldots, m)$ a functional scenario.

We treat the conjunction $P_i \wedge Q_j$ as a functional scenario because it defines an independent function: when P_i is satisfied by the input variables and the initial external variables, the output variables and the final external variables will be defined by Q_j. As highlighted in previous section, since process P may have multiple input and output ports, such a functional scenario may not define an acceptable functional scenario in the sense that the selected input data will lead to the corresponding output result. For this reason, we need to define the notion of *acceptable functional scenario* next.

Definition 3. Let $P_i \wedge Q_j$ be a functional scenario of process P. Then, $P_i \wedge Q_j$ is said acceptable if and only if the following condition holds: $\forall_{in1,in2 \in P_I} \cdot in1 \neq in2 \wedge varSet(P_i) \cap in1 \neq \emptyset \Rightarrow varSet(Q_j) \cap in2 = \emptyset$ where $varSet(P_i)$ denotes the set of free variables occurring in predicate P_i. An acceptable functional scenario $P_i \wedge Q_j$ is expected to ensure that the input satisfying P_i can be used in Q_j to define the output of the process, and therefore requires that P_i and Q_j do not contain input variables of different input ports due to the exclusiveness of input variables of the different input ports in executing process P.

Let us take the process *Charge_Card* as an example to illustrate the notions defined above. The relevant parts of the process are given as follows:

$$Charge\_Card_{pre} = P_1 \vee P_2$$
$$Charge\_Card_{post} = Q_1 \vee Q_2 \vee Q_3 \vee Q_4 \vee Q_5$$

where

$P_1 =$ $amount1 >= 0$

$P_2 =$ $amount2 >= 0$

$Q_1 =$ $railway\_card = modify(\tilde{}railway\_card, balance- > \tilde{}railway\_card.balance + amount1) \wedge account = \tilde{}account \wedge succ\_msg =$ "*Charge is successful*!"

$Q_2 =$ $input\_pass = \tilde{}account.password \wedge input\_acc\_no = \tilde{}account.acc\_no \wedge amount2 <= \tilde{}account.balance \wedge railway\_card = modify(\tilde{}railway\_card, balance- > \tilde{}railway\_card.balance + amount2) \wedge account = modify(\tilde{}account, balance- > \tilde{}account.balance - amount2) \wedge succ\_msg =$ "*Charge is successful*!"

$Q_3 =$ $\neg\ input\_pass = \tilde{}account.password \wedge railway\_card = \tilde{}railway\_card \wedge account = \tilde{}account \wedge unsucc\_msg =$ "*Charge is not successful*!"

$Q_4 =$ $\neg input\_acc\_no = \tilde{}account.acc\_no \wedge railway\_card = \tilde{}railway\_card \wedge account = \tilde{}account \wedge unsucc\_msg =$ "*Charge is not successful*!"

$Q_5 =$ $\neg amount2 <= \tilde{}account.balance \wedge railway\_card = \tilde{}railway\_card \wedge account = \tilde{}account \wedge unsucc\_msg =$ "*Charge is not successful*!"

Functional scenarios	:	(1)	$P_1 \wedge Q_1$
		(2)	$P_1 \wedge Q_2$
		(3)	$P_1 \wedge Q_3$
		(4)	$P_1 \wedge Q_4$
		(5)	$P_1 \wedge Q_5$
		(6)	$P_2 \wedge Q_1$
		(7)	$P_2 \wedge Q_2$
		(8)	$P_2 \wedge Q_3$
		(9)	$P_2 \wedge Q_4$
		(10)	$P_2 \wedge Q_5$
Acceptable functional scenarios	:	(1)	$P_1 \wedge Q_1$
		(2)	$P_2 \wedge Q_2$
		(3)	$P_2 \wedge Q_3$
		(4)	$P_2 \wedge Q_4$
		(5)	$P_2 \wedge Q_5$

Definition 4. We use P_{FS} to denote the set of all possible functional scenarios of process P and P_{AFS} the set of all possible acceptable functional scenarios of process P.

3.2. Animation data generation criteria

Before discussing the criteria for animation data generation, we first need to clarify the basic concepts *animation data* and *animation set* because they are easy to be confused with the notions of test data and test set used in the literature on software testing.

Definition 5. Let P be a process. Let $P_I = \{X_1, X_2, \ldots, X_n\}$, $P_O = \{Y_1, Y_2, \ldots, Y_m\}$, $P_E = \{\tilde{z}_1, \tilde{z}_2, \ldots, \tilde{z}_w, z_1, z_2, \ldots, z_w\}$. Then, an animation data for P, denoted by ad, is a mapping from P_V to the set *Values*:

$ad : P_V \rightarrow Values;$

$ad(v) \in varType(v)$

where

$$P_V = \bigcup_{i \in I} X_i \cup \bigcup_{j \in O} Y_j \cup P_E,\ I = \{1, 2, \ldots, n\},\ O = \{1, 2, \ldots, m\},$$

and

$$Values = \bigcup_{v \in P_V} varType(v).$$

where $varType(v)$ denotes the type of variable v and P_V the union of all of the input variables, the output variables, and the external variables, and *Values* represents the union of all the types of these variables. Note that this definition only explains what an animation data means; it is not used as an animation data generation criterion for our purpose. Animation criteria will be discussed later in this section.

Abstractly speaking, an animation data for an operation is one pair of input and output values. Since an operation usually has more than one input (and output) variable, the input value actually means a set of values for all the input variables, and likewise for the output value. An animation data is usually expressed as a set of pairs of variable with its value, for example, $ad = \{(x_1, 5), \ldots, (\tilde{z}_1, 10), \ldots, (y_1, 50), \ldots, (z_1, 20), \ldots\}$ is a possible animation data for process P, where x_1 denotes an input variable, $\tilde{z}_1$ the initial external variable of a **wr** (writable) external variable z, and y_1 an output variable.

To facilitate our discussions in the rest of this paper, we divide an animation data ad into two parts: (1) input animation data ad_i, which contains values only for the input variables and the initial external variables (e.g., $ad_i = \{(x_1, 5), \ldots, (x_n, 70), (\tilde{z}_1, 10), \ldots, (\tilde{z}_w, 80)\}$), and (2) output animation data ad_o, which contains values only for the output variables and the final external variables (e.g., $ad_o = \{(y_1, 50), \ldots, (y_m, 210), (z_1, 20), \ldots, (z_w, 380)\}$). Such a distinction between input and output animation data will allow us to easily describe the input–output relation for a process in a visualized demonstration as detailed in Section 5.

Definition 6. An animation set for process P is a set of animation data for process P.

We can now define criteria for generating an animation set from a process specification. Each criterion provides a guideline for generating adequate animation set. These criteria are intended to be checked by the tool described in Section 5 when the animation set is produced.

Criterion 1. Let T be an animation set generated from the specification of process P. Then, T must satisfy the following conditions:

(1) $\forall_{Ip \in P_I} \exists_{ad \in T} \cdot ((\forall_{iv \in Ip} \cdot ad(iv) \in varType(iv)) \Rightarrow \exists_{f \in P_{AFS}} \cdot f(ad))$

(2) $\forall_{Op \in P_O} \exists_{ad \in T} \cdot ((\forall_{ov \in Op} \cdot ad(ov) \in varType(ov)) \Rightarrow \exists_{f \in P_{AFS}} \cdot f(ad))$

(3) $\forall_{\tilde{ev}\in P_E}\forall_{ev\in P_E}\exists_{ad\in T}\cdot((ad(\tilde{ev})\in varType(\tilde{ev})\wedge ad(ev)\in varType(ev))\Rightarrow\exists_{f\in P_{AFS}}\cdot f(ad))$

where $ad(iv)$ denotes the value bound for variable iv in animation data ad and $f(ad)$ means that ad satisfies the functional scenario f.

Condition (1) states that for every input variable set Ip there exists an animation data ad in the generated animation set T such that if ad contains a value for every input variable in Ip, ad must satisfy an acceptable functional scenario f. Condition (2) states that for every output variable set Op there exists an animation data ad in T such that if ad contains a value for every output variable in Op, ad must satisfy an acceptable functional scenario f. Condition (3) describes that for every initial external variable $\tilde{ev}$ and final external variable ev there exists an animation data in T such that if ad contains a value for $\tilde{ev}$ and ev, ad must satisfy an acceptable functional scenario f.

This criterion is the least but essential requirement for generating adequate animation set T. Intuitively, it requires that for all of the variables of every input port and output port, an animation data that satisfies some acceptable functional scenario must be generated for animation. It also requires that for all initial and final external variables an animation data satisfying some acceptable functional scenario must be generated. Thus, the criterion ensures that all of the variable groups involved in the process specification are animated at least once, respectively.

Note that if no input could be produced to satisfy the pre-condition of process P, then animation of process P will not be conducted. This, however, does not necessarily mean that the process specification is semantically wrong, but indicates that the process cannot be properly used in the system. Another possibility is that when an input is successfully generated to satisfy the pre-condition, no corresponding output can be produced to satisfy the post-condition of the process. In this case, no successful animation data is generated, which might indicate a potential fault in the post-condition. Humans involved must check the specification to decide the root of the problem and what to do with it.

For instance, an animation set containing the following two animation data for process *Charge_Card* satisfies this criterion:

$$
\begin{aligned}
ad^1 &= \{(amount1 = 10),\\
&\quad (succ\_msg = \text{“}Charge\ is\ successful!\text{”}),\\
&\quad (\tilde{railway\_card} = \{\text{“}J.Smith\text{”}, 1000\}),\\
&\quad (railway\_card = \{\text{“}J.Smith\text{”}, 1010\}),\\
&\quad (\tilde{account} = \{[2,5,9,1,6,3,2,4,8],\\
&\quad [1,2,5,3,9,5], 1500),\\
&\quad (account = \{[2,5,9,1,6,3,2,4,8],\\
&\quad [1,2,5,3,9,5], 1500)\}\\
ad^1_i &= \{(amount1 = 10),\\
&\quad (\tilde{railway\_card} = \{\text{“}J.Smith\text{”}, 1000\}),\\
&\quad (\tilde{account} =\\
&\quad \{[2,5,9,1,6,3,2,4,8], [1,2,5,3,9,5], 1500)\}\\
ad^1_o &= \{(succ\_msg = \text{“}Charge\ is\ successful!\text{”}),\\
&\quad (railway\_card = \{\text{“}J.Smith\text{”}, 1010\})\\
&\quad (account = \{[2,5,9,1,6,3,2,4,8],\\
&\quad [1,2,5,3,9,5], 1500)\}
\end{aligned}
$$

$$
\begin{aligned}
ad^2 &= \{(amount2, 5000),\\
&\quad (input\_pass, [1,2,5,3,9,5]),\\
&\quad (input\_acc\_no, [2,5,9,1,6,3,2,4,8]),\\
&\quad (unsucc\_msg, \text{“}Charge\ is\ not\ successful!\text{”}),\\
&\quad (\tilde{railway\_card} = \{\text{“}J.Smith\text{”}, 1000\}),\\
&\quad (\tilde{account} = \{[2,5,9,1,6,3,2,4,8],\\
&\quad [1,2,5,3,9,5], 1500),\\
&\quad (railway\_card = \{\text{“}J.Smith\text{”}, 1000\}),\\
&\quad (account = \{[2,5,9,1,6,3,2,4,8],\\
&\quad [1,2,5,3,9,5], 1500)\}\\
ad^2_i &= \{(amount2 = 5000),\\
&\quad (input\_pass = [1,2,5,3,9,5]),\\
&\quad (input\_acc\_no = [2,5,9,1,6,3,2,4,8])\},\\
&\quad (\tilde{railway\_card} = \{\text{“}J.Smith\text{”}, 1000\}),\\
&\quad (\tilde{account} =\\
&\quad \{[2,5,9,1,6,3,2,4,8], [1,2,5,3,9,5], 1500)\}\\
ad^2_o &= \{(unsucc\_msg = \text{“}Charge\ is\ not\ successful!\text{”}),\\
&\quad (railway\_card = \{\text{“}J.Smith\text{”}, 1000\}),\\
&\quad (account =\\
&\quad \{[2,5,9,1,6,3,2,4,8], [1,2,5,3,9,5], 1500)\}
\end{aligned}
$$

Using these two animation data, we can perform two animations of the process *Charge_Card*. One is a visualized demonstration of the action of inputting $amount1 = 10$ and producing the output $succ\_msg$ = “*Charge is successful*!” and updating the balance of the store *railway_card* but keeping the *account* unchanged. Fig. 3 shows the resultant state of the animation.

Another is a visualized demonstration of providing the inputs $amount2 = 5$, $input\_pass = [1,2,5,3,9,5]$, and $input\_acc\_no = [2,5,9,1,6,3,2,4,8]$, and producing the output $unsucc\_msg$ = “*Charge is not successful*!” without updating any of the stores *railway_card* and *account*. The resultant state of this animation is illustrated in Fig. 4.

Due to the fact that the input ports or output ports do not have a one-to-one relation with the acceptable functional scenarios derived from the pre- and post-conditions, this criterion apparently does not ensure that every acceptable functional scenario is animated. For this reason, we form another criterion.

Criterion 2. Let T be an animation set generated from the specification of process P that satisfies Criterion 1. Then, T must satisfy the following condition: $\forall_{f\in P_{AFS}}\exists_{ad_1\in T}\cdot f(ad_1)$

Intuitively, this criterion requires that every acceptable functional scenario be animated with at least one animation data in animation set T to check its validity. This is reasonable because every acceptable functional scenario is expected to define a desirable behavior of the process with respect to the user's requirements.

However, due to the possibility of human mistakes, some of the unacceptable functional scenarios of the process might be improperly defined (e.g., using wrong terms or operators defined on types). For this reason, we define another criterion to allow every possible functional scenario to be animated at least once.

Criterion 3. Let T be an animation set generated from the specification of process P that satisfies Criterion 2. Then, T must satisfy the following condition:

$\forall_{f\in P_{FS}}\exists_{ad_1\in T}\cdot f(ad_1)$

This criterion allows to cover all of the functional scenarios in animation for different purposes. For the acceptable functional scenarios, animation is intended to check its validity, but for the unacceptable functional scenarios, animation is usually intended to confirm that they are indeed undesirable behaviors. This is similar to program testing where normal inputs are used to

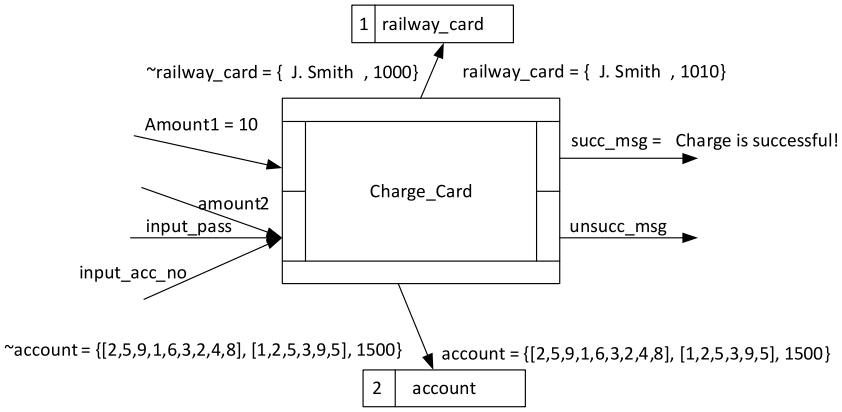

Fig. 3. Animation of Charge_Card with animation data.

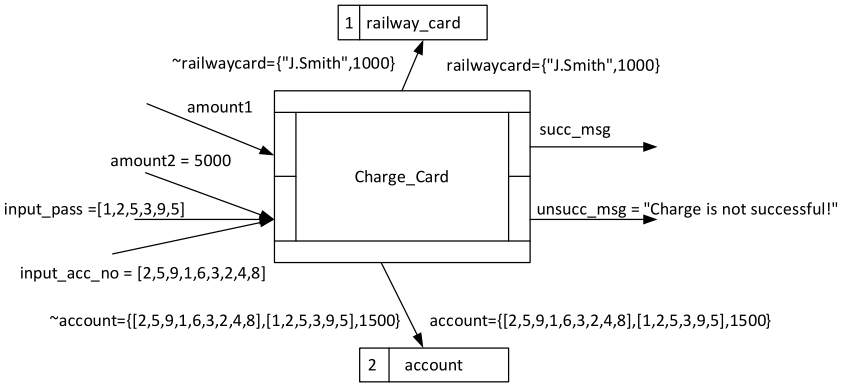

Fig. 4. Animation of Charge Card with data ad^2.

confirm whether the expected behavior of the program is provided correctly but some exceptional inputs may also be used to check whether the program properly deals with the situation, for example, by issuing an error message. The difference is that in the case of program testing, the program needs to give an appropriate response but in the case of specification animation, the humans (e.g., the developer or the user or both) need to interpret and determine the nature of the situation. Note that it is possible to generate an animation data involving input variables of different input ports that satisfies an unacceptable functional scenario because all of the input variables involved can be assigned a specific value from its type that satisfies the conditions of the functional scenario. However, in this case, since the exclusiveness of different input ports, as part of the actual pre-condition of the process, is violated, even if the functional scenario is satisfied, it has nothing to do with the correctness of the specification, but may provide a chance to let the analyst think about this exceptional case. Perhaps he or she may realize that putting some of the involved input variables into different input ports is a mistake through this animation, and therefore correct the mistake by changing the structure of the process and its specification.

4. Animation data generation algorithms

To validate a process using animation, we must carry out an animation for every acceptable functional scenario of the process in order to ensure that each potentially valid functional behavior is demonstrated at least once. To this end, animation data generation must focus on each individual functional scenario $P_i \wedge Q_j$ where P_i only contains input variables whilst Q_j contains output variables (possibly input variables as well). Apparently, as long as we generate an animation data satisfying this functional scenario, we can use the animation data to demonstrate the input–output relation of the process for validation. The demonstration of the input–output relation can be visualized with a tool support, as discussed later in Section 5.

Since each functional scenario $P_i \wedge Q_j$ is a conjunction of atomic predicates where each atomic predicate is a relation or its negation, generation of animation data to satisfy the scenario must involve the animation data generation for atomic predicates and their conjunction, respectively. For this reason, we first discuss animation data generation from atomic predicates and then extend it to generation from conjunctions.

4.1. Animation data generation from atomic predicates

For the sake of simplicity in the discussions below, we represent the scenario $P_i \wedge Q_j$ as the conjunction: $A_1 \wedge A_2 \wedge \cdots \wedge A_m$, where each $A_k(x_1, x_2, \ldots, x_q)$ $(k = 1, \ldots, m, q \geq 1)$ denotes an atomic predicate. The variables $x_1, x_2, \ldots, x_q$ are free variables, which can be either input variables or output variables of the related process, but may be part of all the free variables $x_1, x_2, \ldots, x_w$ used in the whole conjunction, where $w \geq q$. A variable can denote either a numeric value or a value of compound type (e.g., set, sequence, map, or composite type) that is available in many of the model-based formal notations, such as VDM-SL, Z, and SOFL (Liu, 2004), although their syntax may differ slightly. We start discussing the case of atomic predicates with only numeric variables, and then extend the discussion to predicates with variables of compound types.

4.1.1. Numeric type variables

An atomic predicate usually takes the format $E_1 \ominus E_2$ where E_1 and E_2 are both algebraic expressions possibly involving all the

free variables x_1, x_2, ..., x_q, and $\ominus \in \{=, >, <, \geq, \leq, \neq\}$ is a relational operator. An algorithm to generate an animation data satisfying the predicate takes the following steps:

Step 1: Randomly generate a value v_i for each variable x_i ($i = 2, 3, \ldots, q$) within its type and substitute the value for the variable in $E_1 \ominus E_2$;

Step 2: Convert $E_1 \ominus E_2$ into an equivalent predicate $x_1 \ominus C$ according to algebraic laws, where C is an constant;

Step 3: Generate a value v_1 for x_1 such that $v_1 \ominus C$ holds;

Step 4: Treat $v_1, v_2, \ldots, v_q$ as an animation data.

Although this algorithm is rather straightforward, an example will help the reader comprehend its essential idea even better. Consider the atomic predicate $x + y * z < x * y + z$. The steps taken to generate an animation data to satisfy the predicate are as follows:

Step 1: Randomly generate values for y and z, such as $y = 8$ and $z = 10$. Thus, we obtain $x + 8 * 10 < x * 8 + 10$.

Step 2: Convert $x+8*10 < x*8+10$ into the equivalent predicate $10 < x$.

Step 3: Generate a value 11 for x.

Step 4: Treat 11, 8, and 10 for x, y, and z, respectively, as an animation data that satisfies $x + y * z < x * y + z$.

4.1.2. Compound type variables

The above algorithms cannot directly handle variables of compound data types due to the possible use of the operators defined on those types. Since the essential idea for handling operations for all kinds of compound types is similar, we only choose the *map type* in SOFL as an example to discuss the algorithms for animation data generation. The algorithms for dealing with the other compound types in the SOFL language, such as set types, sequence types, and composite types, have been discussed in our previous publication (Liu and Nakajima, 2010).

A map is a mathematical function, describing a finite association between the domain and the range of the map. It is usually represented by a set of pairs, such as $m = \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}$. In each pair $a_i \rightarrow b_i$ ($i = 1, 2, \ldots, n$), a_i and b_i are called *domain element* and *range element* of the pair, respectively. There are several operators defined on map types, such as *dom* (domain), *rng* (range), *map application*, *domrt* (domain restriction to), *rngrt* (range restriction to), *domrb* (domain restriction by), *rngrb* (range restriction by), *override* (map override), *inverse* (map inverse), *comp* (map composition), $=$ (equality) and $\langle\rangle$ (inequality). Let m, m_1 and m_2 be a map, respectively, from *int* to *int*, then, all of the operators are interpreted as follows:

- $dom(m)$: the domain of map m.
- $rng(m)$: the range of map m.
- $m(a)$: the range element associated with domain element a in m.
- $domrt(s, m)$: the sub-map of m that contains all of the pairs whose domain element falls into set s.
- $rngrt(m, s)$: the sub-map of m that contains all of the pairs whose range element falls into set s.
- $domrb(s, m)$: the sub-map of m that contains all of the pairs whose domain element does not fall into set s.
- $rngrb(m, s)$: the sub-map of m that contains all of the pairs whose range element does not fall into set s.

Table 1
Algorithms for animation data generation from map type expressions.

No.	Expressions	Algorithms for animation data generation
(1)	$dom(m) = s$	$m := \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}$, assuming $s = \{a_1, a_2, \ldots, a_n\}$ and $b_k \in int$ $(k = 1, 2, \ldots, n)$
(2)	$rang(m) = s$	$m := \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}$, assuming $s = \{b_1, b_2, \ldots, b_n\}$ and $a_k \in int$ $(k = 1, 2, \ldots, n)$
(3)	$m(a_1) = b_1$	$m := \{a_1 \rightarrow b_1\}$
(4)	$domrt(s, m) = m_1$	$m := \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_k \rightarrow b_k\}$, assuming $s = \{a_1, a_2, \ldots, a_k\}$ and $m_1 = \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}, n \geq k$
(5)	$rngrt(m, s) = m_1$	$m := \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_k \rightarrow b_k\}$, assuming $s = \{b_1, b_2, \ldots, b_k\}$ and $m_1 = \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}, n \geq k$
(6)	$domrb(s, m) = m_1$	$m := \{a_{k+1} \rightarrow b_{k+1}, a_{k+2} \rightarrow b_{k+2}, \ldots, a_n \rightarrow b_n\}$, assuming $s = \{a_1, a_2, \ldots, a_k\}$ and $m_1 = \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}, n \geq k$

Table 2
Table caption.

No.	Expressions	Algorithms for animation data generation
(7)	$rngrb(m, s) = m_1$	$m := \{a_{k+1} \rightarrow b_{k+1}, a_{k+2} \rightarrow b_{k+2}, \ldots, a_n \rightarrow b_n\}$, assuming $s = \{b_1, b_2, \ldots, b_k\}$ and $m_1 = \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}, n \geq k$
(8)	$inverse(m) = m_1$	$m := \{b_1 \rightarrow a_1, b_2 \rightarrow a_2, \ldots, b_n \rightarrow a_n\}$, assuming $m_1 = \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}$
(9)	$comp(m_1, m_2) = m$	$m_1 := \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}$, $m_2 := \{b_1 \rightarrow c_1, b_2 \rightarrow c_2, \ldots, b_n \rightarrow c_n\}$, assuming $b_i \in int$, $i = 1, 2, \ldots, n$, and $m = \{a_1 \rightarrow c_1, a_2 \rightarrow c_2, \ldots, a_n \rightarrow c_n\}$
(10)	$m_1 = m_2$	$m_1 := \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}$, $m_2 := \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}$
(11)	$m_1 <> m_2$	$m_1 := \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_n \rightarrow b_n\}$, $m_2 := \{a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots, a_k \rightarrow b_k\}$, where $k < n$.

- $override(m_1, m_2)$: the map obtained by taking all of the pairs in m_2 and all of the pairs in m_1 whose domain element is not the same as that of any pairs in m_2.
- $inverse(m)$: the map obtained by exchanging the domain element with the range element of all the pairs in m. This can be applied only when m defines a one-to-one association from the domain to the range.
- $comp(m_1, m_2)$: the map obtained by composing map m_1 and m_2.
- $m_1 = m_2$: evaluates to true if m_1 is the same as m_2; otherwise, evaluates to false.
- $m_1 \langle\rangle m_2$: evaluates to true if m_1 is different from m_2; otherwise, evaluates to false.

Tables 1 and 2 describe our proposed algorithms to generate animation data for maps m, m_1, and m_2 from the relevant expressions involving the operators just mentioned above. In the tables, we use $m := E$ to mean that m is generated as E, where E can be any appropriate expression.

It is worth mentioning that the algorithms in Tables 1 and 2 only show one possibility in animation data generation. There are many other possibilities for the generation algorithms. It is important for the developer in charge of the specification animation to take a flexible approach in adopting data generation algorithms based on our proposals.

4.2. Generation from conjunctions

We can now focus on animation data generation from the conjunction $A_1 \wedge A_2 \wedge \cdots \wedge A_m$.

4.2.1. Simple algorithm

A simple algorithm for the purpose is first to generate an animation data satisfying one of the atomic predicates, say A_i $(i = 1, \ldots, m)$, and then use the same animation data to evaluate the rest predicates in the conjunction. If it also satisfies all of the rest predicates, a qualified animation data for the conjunction is found; otherwise, another attempt to generate a new animation data must be made to repeat the same process. However, our experience suggests that this algorithm may not be efficient in many situations.

Existing SAT solvers, such as RISS (Manthey, 2012), may be used for animation data generation for a conjunction, but since the SAT solver only deals with propositional logic, its capability is limited for our formal notation that adopts first-order predicate logic with rich data types. Existing SMT solvers, such as Yices (Dutertre, 2014) and Z3 (Moura and Bjorner, 2008), can be a better possibility for the solution due to their capability of dealing with predicate logic. However, in the SOFL language, an expression may involve operators defined on some types (e.g., set, sequence, map types), and those operators (e.g., *domrb*(s, m)) are not directly dealt with by the SAT or SMT solver. We have tried hard to adopt Z3 for the generation of animation data in our work, but could not succeed because of its limitation in dealing with many operators defined on compound data types in the SOFL language. To overcome this difficulty, we propose a more efficient algorithm than the simple one described above for animation data generation from a logical conjunction.

4.2.2. More efficient algorithm

In this section, we describe an algorithm for generating animation data from a conjunction that is more efficient than the simple algorithm mentioned above. The essential idea of the algorithm is first to form an *ordered partition* of the atomic predicate set $\{A_1, A_2, \ldots, A_m\}$ according to *variable dependency*, and then properly apply the simple algorithm mentioned above to generate a qualified animation set for the conjunction if it is satisfiable. Before introducing the details of the algorithm, we first need to introduce the notions to be used in the algorithm below.

Definition 7. If predicate E_2 contains more free variables than predicate E_1, denoted by $VSet(E_1) \subset VSet(E_2)$ where $VSet(E_1)$ denotes the set of all free variables occurring in expression E_1, then we say that E_2 is dependent on E_1, represented by $E_1 \sqsubset E_2$.

For example, predicate $a * b - d > 50$ is dependent on $d + b > 10$; that is, $d + b > 10 \sqsubset a * b - d > 50$, because the former contains three variables a, b, and d whilst the latter contains two variables d and b.

Definition 8. The set of predicate sets $\{P_1, P_2, \ldots, P_w\}$ is an ordered set of predicate sets on $\sqsubset$ if it satisfies the following two conditions:

(1) $P_i \sqsubset P_{i+1}$ where $i \geq 1 \wedge i \leq w - 1$,

(2) No predicate in any of the predicate sets is dependent on another predicate in the same predicate set.

In this definition, $P_i \sqsubset P_{i+1}$ means that any predicate in P_{i+1} is dependent on every predicate in P_i.

Definition 9. An animation data ad is said to satisfy a predicate set R if it satisfies every predicate in R.

Algorithm 4.2.1. /*Java-based pseudocode*/

No. 1 Construct a partition $\{P_1, P_2, \ldots, P_w\}$ for the conjunction $A_1 \wedge A_2 \wedge \cdots \wedge A_m$ $(1 \leq w \leq m)$ such that $\{P_1, P_2, \ldots, P_w\}$ forms an ordered set of predicate sets on $\sqsubset$;

No. 2 $ad_0 := \{\}$; $i := 1$; $flag := 0$; /*initializing variable ad_0 representing the initial animation data*/

No. 3 *while* ($i > 0$ && $i \leq w$ && $flag \leq NoOfFailure$) {

$Q := ObtainInstantiatedPredicates(P_i, ad_{i-1})$;

/*Q is an array of predicates*/

$ad_i := GenerateAnimationData(Q)$;

/*ad_i is a new animation data generated based on the predicates in Q*/

if $(ad_i = \{\})$

$\{i := i - 1;\ flag := flag + 1;\}$

else $\{i := i + 1;\}$

}

No.4 *if* $(i <= 0 \parallel flag > NoOfFailure)$ {Display an animation data generation failure message}

else {Display an animation data generation success message and ad_i

is the generated animation data}

No. 5 End.

This algorithm aims to produce an animation data that satisfies all of the predicates in P_i $(1 \leq i \leq w)$ and then utilize the values in the animation data to generate a more complete animation data for P_{i+1}. Repeat this process until P_w is reached and a qualified animation data is generated. However, if the generation fails for P_i, it will go one step back to retry generating an animation data for P_{i-1} and then repeat the same process until reaching the level that causes the problem. But if the number of failures to generate the qualified animation data satisfying all P_1, P_2, ..., P_w reaches the pre-defined number denoted by *NoOfFailure*, or no animation data can be generated for P_1, a failure message will be issued as the result of the algorithm.

In the algorithm, the *function ObtainInstantiatedPredicates*(P_i, ad_{i-1}) obtains an array Q whose elements are the atomic predicates resulting from substituting the value of every variable in animation data ad_{i-1} for the same variable in the atomic predicates in P_i. For instance, suppose

$P_i = \{x * y + z > 1, x + y * z < 100\}$, containing two predicates and

$P_{i-1} = \{x + y > 10\}$, and the animation data for P_{i-1} is

$ad_{i-1} = \{(x, 8), (y, 9)\}$. Then, we get Q from *ObtainInstantiated Predicates*(P_i, ad_{i-1}) as follows:

$Q = \{8 * 9 + z > 1, 8 + 9 * z < 100\}$.

To generate an animation data for P_i based on Q, we need to apply the function *GenerateAnimationData*(Q). The animation data generated from this function is actually a more complete one than ad_{i-1} that satisfies all of the atomic predicates in Q. Assume that array Q has n atomic predicates as its elements, we give an algorithm used to implement the function *GenerateAnimationData*(Q) below.

Algorithm 4.2.2. /* Java-based pseudocode*/

```
satisfyingConjunction:= false;
for (int i := Q.length() − 1; i >= 0; i − −){
ad_c := GenerateAnimationDatafromPredicate(Q[i]);
j := Q.length() − 1;
while (j >= 0 && Satisfy(Q[j], ad_c)){
        j := j − 1;
      }
  if (j >= 0)
     {Q := Rotate(Q); }
  else {satisfyingConjunction := true;
        break;}
    }
  if (satisfyingConjunction == false) {
    ad_c = {};
   }
 return ad_c;
```

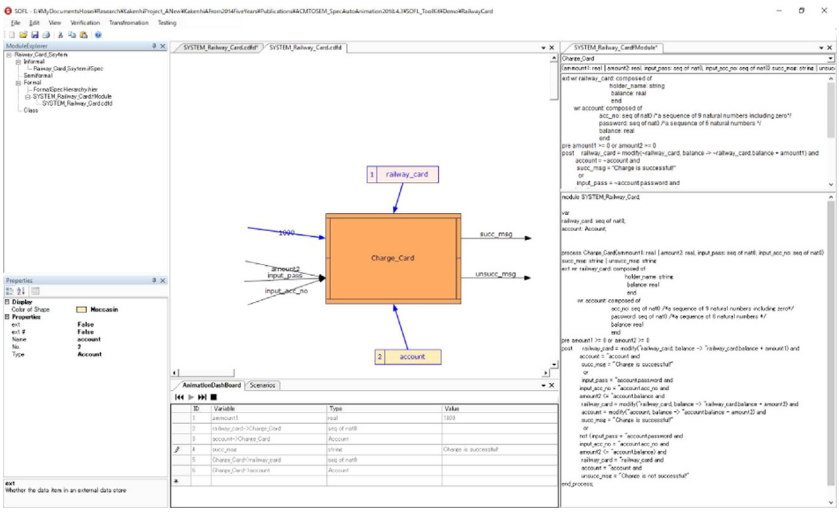

Fig. 5. Snapshot of the animation of process *Charge_Card*.

This algorithm first tries to generate an animation data satisfying the last atomic predicate of Q and then to test whether it satisfies all of the other atomic predicates in Q. If yes, a successful animation data is generated; otherwise, repeat the same process for the other atomic predicates in Q until all of the atomic predicates of Q is exhausted. In this algorithm, the function *GenerateAnimationDatafromPredicate*($Q[i]$) produces an animation data as the result that satisfies the i*th* atomic predicate in Q. *Satisfy*($Q[j], ad_c$) yields true if animation data ad_c satisfies $Q[j]$; otherwise it yields false. *Rotate*(Q) yields a permutation of Q by moving the last element of Q to the first position and all of the other elements are shifted one position to the right in Q.

Note that the algorithms for generating animation data above work well generally, but no guarantee can be given to ensure that an animation data satisfying the conjunction will always be found or will always be efficiently found. If the generation of an animation data is unsuccessful, human must check and determine whether there exists any data satisfying the conjunction.

5. Animation tool

We have developed a prototype tool to support the animation of a process specification. In addition to supporting animation data generation based on the criteria and algorithms discussed above, the tool mainly offers three animation functions: (1) input–output relation animation, (2) data animation, and (3) logical expression animation. The tool is developed using C# in the Visual Studio 2012 environment.

5.1. Input–output relation animation

An important step in process specification animation is visualized demonstration of the input–output relation defined in the process specification. To enable the demonstration to effectively help humans (analyst or user or both) make judgments on the validity of the specification, our tool supports the input–output relation demonstration.

Before demonstrating the input–output relation of the process, the tool facilitates the user to choose a functional scenario and allows animation data to be generated both automatically and manually. Once an animation data is made available, by clicking on the right button, the tool will start to show how the selected input data flows are used to produce the expected output data flows. If data stores are connected to the process under animation, the access or updating of the stores is also properly demonstrated. To attract the user's attention, the tool may choose different colors to represent the input and output data flows, respectively.

Fig. 5 shows a snapshot of visualized demonstration of the process *Charge_Card*. The process is represented graphically in the middle pane of the GUI and its formal specification is given in the right pane. When the input data flow *amount*1 is selected, its animation data can be automatically generated and presented in the area below the pane. The animation data can also be provided by the user if such an option is chosen. When the arrow button at the left-bottom of the middle pane is clicked, a visualized demonstration of the behavior of updating the store *railway_card* and producing the value of the output data flow variable *succ_msg* (i.e., "Charge is successful!") from the value of the input data flow variable *amount*1 will be performed comprehensibly. The visualized demonstration can be done either completely automatically or step by step with the user's operation.

5.2. Data animation

When performing animations for a process, the user (or client) might not understand the characteristics of the input or output data, especially when the data structure is complex, such as a set, sequence, composite object, or map. To help the user learn about the data, data animation is provided by a sub-system of the tool with the aim of explaining the characteristics of the data in a comprehensible manner.

For example, when the selected data is a set of natural numbers, the data animation will display a moving pictures of showing the following three things: (1) each element of the set is represented by a "ball" and all of the elements are being put into a "container" (representing the set), as illustrated by the snapshot of the tool in Fig. 6, (2) the order of the elements occurring in the set is not important, and (3) no element duplication is allowed in the set. For item (2), the tool shows dynamic changes of the position of some elements in the container, and for item (3), the tool automatically generates a small set of elements of the same type and shows that the elements, one by one, can be put into the container if it is not a member of the selected set, or cannot be put into the container if it is already a member of the set. The snapshot of the tool in Fig. 7 shows one scene of the animation process.

Moreover, to help the user more efficiently understand the nature of the set, the tool also offers voice explanation (i.e., speak loudly) of the above three things as the animation is performing.

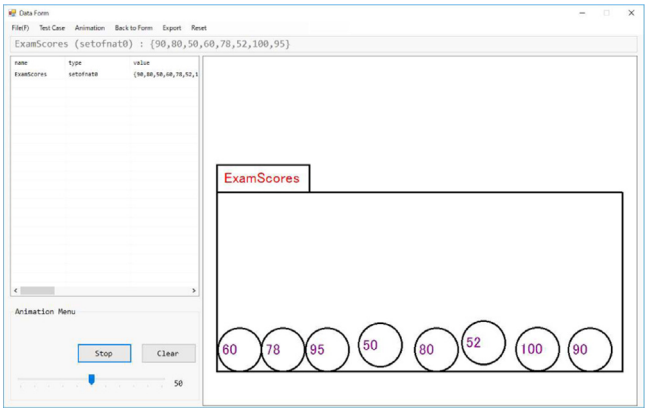

Fig. 6. Graphical representation of a set in animation.

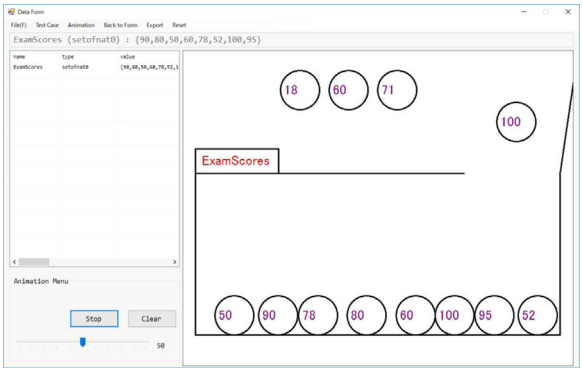

Fig. 7. Set characteristics animation.

Table 3
The summary of data animation.

Type	Characteristics
Set types	(1) The order of element occurrence is not important. (2) No element duplication is allowed.
Sequence types	(1) The position of element occurrence is significant. (2) Element duplication is possible.
Composite types	(1) An object can have multiple fields possibly with different types.
Map types	(1) For any one element in the domain, there is at most one associated element in the range. (2) Both the domain and the range are finite.

This function is implemented by utilizing the Microsoft *SpeechLib* package that provides sufficient classes and methods to transform texts into voice reading the text out loud. The voice description together with the moving animation produces the similar effect to that of reading with listening in English study as a non-native speaker or that of listening to the voice instruction from a car navigation system in driving. Unfortunately, despite the fact that the voice feature is consistent with the general principle of enhancing human understanding in software engineering, its effect is extremely difficult, if not impossible, to measure due to the uncertain character of human judgment.

Similarly, we also deal with other compound data types available in SOFL, such as sequence, composite, and map types. For each type, the animation properly shows its characteristics. For brevity, we omit the detailed discussions on each of the type animation, but give a summary of all the type animations in Table 3 for reference.

The animation actions are implemented using the *System.Drawing* and *System.Threading* packages available in the Visual Studio 2012 environment. The voice description is automatically generated for the selected data type.

5.3. Logical expression animation

Our experience with industrial collaboration suggests that the analyst and/or the user sometimes may be interested in some predicate expressions or individual operators defined on some data types (e.g., set types, map types) during a process specification animation. To help them check the appropriateness of the predicate expressions or individual operators, the tool also supports animations of predicate expressions and the most frequently used but perhaps complicated operators defined on all of the four compound types mentioned in Table 3. The question is how to perform the animation of a predicate expression to allow the user to easily comprehend its meaning.

The animation of logical expressions can be done at two levels. One is for a functional scenario form (FSF), which is a disjunction of functional scenarios, of the process under consideration and the other is for a single atomic predicate or even a single operator defined in the SOFL language.

The animation of an FSF is carried out in two steps. The first step is to convert the FSF into a finite state machine, and the second step is to evaluate the states along the transitions between the states using animation data. In the finite state machine, each state represents an atomic predicate involved in some functional scenario and the transition from one state to another denotes the logical operator “and”. Therefore, each functional scenario is represented by a sequence of state transitions from the start state to an end state. This sequence actually represents graphically the sequential steps of evaluating the atomic predicates involved in the corresponding functional scenario. For this reason, by demonstrating the sequential steps, we will be able to see how each functional scenario is actually evaluated to true or false with a successful animation data and a failed animation data, respectively. Thus, the contents of each functional scenario can be analyzed by the developer in a relatively comprehensible manner. The whole state machine is described as a set of sequences of state transitions from the start state to the end states. For example, suppose a functional scenario form (FSF) is $Q1 \wedge Q2 \wedge Q3 \vee P1 \wedge P2 \vee W1 \wedge W2 \wedge W3$ and each atomic predicate (e.g., $Q1$, $P1$, $W1$) involves three free integer variables x, y, and z. Then, a finite state machine for this FSF is constructed as shown in Fig. 8, where S denotes the start state and E denotes the end state. An animation of a functional scenario in this FSF starts from the start state S and go through one of the transition sequence of the three transition sequences shown in the figure to reach the end state E. Each transition from state $Q1$ to state $Q2$, for example, means that the atomic predicate $Q1$ evaluates to true and the next evaluation will be the atomic predicate $Q2$.

As far as the animation of an atomic predicate is concerned, our focus is on the explanation of its meaning in a comprehensible manner. Let us use the atomic predicate x *inset* X as an example to explain how the animation is performed, where x denotes a value in the natural number type $nat0$ (including zero) and X is a subset of $nat0$. This predicate evaluates to true if x is a member of set X; otherwise, it evaluates to false. To perform the animation, the tool first requests the user of the tool to select the type of x and to supply specific values for x and X, respectively. Then, the tool will automatically display three things in turn: (1) the process of all of the set elements being put into a container, (2) the atomic predicate with specific values for x and X, and (3) the confirmation whether the value for x is a member of the set and the evaluation result of the predicate is consistent with the

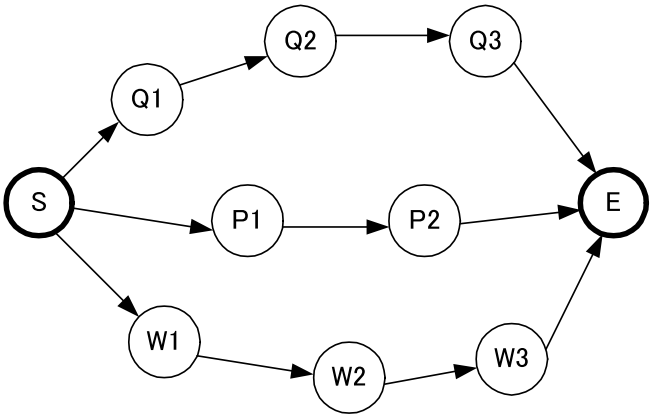

Fig. 8. An illustration of animation for an FSF.

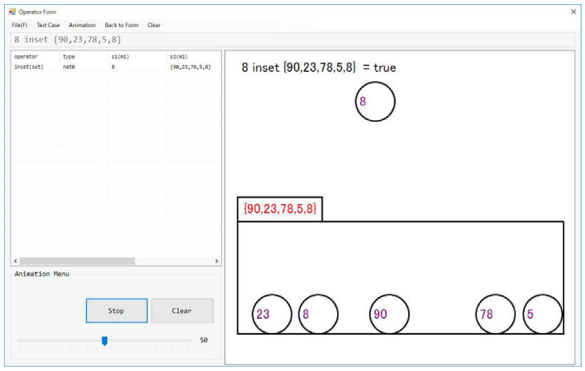

Fig. 9. Snapshot of logical expression animation.

confirmation result. Fig. 9 shows the snapshot of a scene in the predicate animation.

As far as animation for individual operators is concerned, we have also provided comprehensible animation for other operators adopted in SOFL. Table 4 gives a summary of animation of all of the individual operators defined on various compound data types in SOFL. The important thing of our work here is to show the essential idea of whether and how animation can be conducted, which can be applied or extended to deal with animation of similar formal notations.

In fact, our experience suggests that doing animation for validation of a formal model be actually a process of "educating" the humans (e.g., the end user or the analyst) what the model does and what kind of data are used. Only after the humans get a precise understanding of the features of the data and the logical expressions involved, they can make a fair judgment on what is correctly built and what is not with respect to their desires. Apparently, if the humans are well trained in the formal notation, they may benefit less from the tool than those who are unfamiliar with the formal notation.

6. Application in practice

As is well-known, conducting a credible experiment for evaluation of software engineering techniques is notoriously difficult due to the fact that many uncertain factors may prevent us from setting up a context in which our method is comparable with any other method. For this reason, we do not believe that using controlled experiments is definitely a credible way to evaluate software engineering techniques within a limited resources (e.g., time, budget, subjects), especially when human operations and judgments are involved. Instead, we believe that as long as a technique like ours is confirmed to be effective in practice by some practitioners, even in a single company, the result will be valuable in the sense that it indicates a high possibility that the technique may also be applicable to other software systems by other practitioners in the future. This idea seems to be consistent with that of *trial experiment* and *comparison experiment* described in Tedre and Moisseinen (2014). To this end, we have applied our animation technique to an Automated Train Protection (ATP) software development project for a railway signal company in China. ATP software is one of the kernel components of railway transportation system, which performs speed control functionality of a train. As a typical safety-critical system, the ATP software requirements must be carefully validated to let ATP users confirm whether their perceptions are completely and correctly defined in the formal specification.

Table 4
Operators defined on data types in SOFL.

Type	Atomic predicate or Operator	Meaning
Set type	x *inset* X x *notin* X $card(X)$	membership non-membership cardinality
	$union(X, Y)$	union
	$inter(X, Y)$	intersection
	$diff(X, Y)$	difference
	$subset(X, Y)$	subset
	$psubset(X, Y)$	proper subset
	$power(X)$	power set
Sequence type	$hd(a)$	head
	$tl(a)$	tail
	$len(a)$	length
	$elems(a)$	element set
	$inds(a)$	index set
	$a(i)$	sequence application
	$conc(a, b)$	concatenation
Composite type	$c.f$	selecting field
	$modify(c, f \rightarrow v)$	field modification
Map type	$m(a)$	map application
	$dom(m)$	domain of m
	$rng(m)$	rang of m
	$domrt(s, m)$	domain restriction to
	$rngrt(m, s)$	rang restriction to
	$comp(m_1, m_2)$	composition
	$override(m_1, m_2)$	override

6.1. Application procedure

We applied our method to animate the formal specification of the ATP software system for validation. The formal specification was composed of 417 processes and constructed by the requirements analysts of the company using the SOFL language. We generate animation data using our method and then carry out the animation for all the processes in the specification.

Before applying our method, two experienced requirements engineers in the company were asked to manually review the specification. These two engineers joined our study by serving as the end-user of the system to help judge whether each animation

Table 5
The results of animation and review.

Methods	Objects of validation	Detected faults	Time consumed
Animation	Functionality	29	80 h
	Data items	26	
	Logical expressions	67	
Review	Functionality	5	200 h
	Data items	7	
	Logical expressions	15	

finds defects in the specification. Thus, the two engineers can tell us the difference between the defects found by the animation and those found by the previous review.

We took three steps to complete our animation. Firstly, we performed the animation of the functionality of the processes involved. During this animation, the focus is put on the analysis of the input–output relationships. Secondly, we carry out data animation for variables of compound types (e.g., set types, sequence types, map types) identified in the specifications of the processes. Since variables of basic types (e.g., numeric types) are easy to comprehend, we did not carry out the animation for them. Finally, we performed the animation of the complex logical expressions extracted from the specifications of the processes. The simple logical expressions are not chosen for animation since they can be easily checked by a simple review.

6.2. Results of application

Table 5 shows the result of applying our animation method to check the functionality of all 417 processes and the 252 data items of compound types and the 33 complex logical expressions extracted from the process specifications. In comparison, the table also shows the result of reviewing all the process specifications and the corresponding data items and logical expressions.

The result of the study shows a considerable effectiveness of our animation approach in comparison with the specification review in terms of fault detection. Supported by the animation tool, the requirements analysts found 122 faults (29 function-related faults, 26 data-related faults, and 67 logic-related faults) in the formal specification within 80 h in 10 working days. In comparison with the effect of the specification review conducted before the animation, our animation approach finds 24 more function-related faults, 19 more data-related faults, and 52 more logic-related faults than the review approach, and saves approximately 120 h time. It is confirmed that all of the faults found by the review are identified by the animation, but it is not necessarily true the other way round. Moreover, since the animation was done with the tool support and the review was done manually without a specialized tool support, the animation took much less time than the review.

We could also get more reliable feedback from the engineers about the effect of the visualization and voice explanation of our animation technique if more engineers could be used. But unfortunately, this could not be done due to the constraints on the resources. We will try to conduct more mature empirical studies in the future when necessary resources become available.

6.3. Limitations

Through the application of the tool-supported approach in practice, we have also found some limitations. The first one lies in the subjective evaluation of the validity of the specification. Although the animation can intuitively show the system functionality, determining whether the specification conforms to the user's intention, to some extent, still depends on human's experience and understanding of the requirements and domain knowledge. Therefore, some faults that are related to domain knowledge may not be effectively detected by the tool. The second major limitation lies in the construction of formal specification. The animation requires precise definition of data types and functions. That is, to use the tool-supported animation approach, formal specifications need to be constructed. However, in most industrial enterprises, requirements are not fully formalized, rather, they are described in certain informal or semi-formal manner (i.e., the UML). Currently, only in some safety-critical domains, such as the railway control, the aviation and the spacecraft domains, formal specifications are likely to be written. For this reason, the proposed animation technique is currently limited to the applications in these critical domains.

In spite of these limitations, the application of the animation technique in the real industrial project has given us the confidence that it can help engineers improve requirements specifications in practice.

7. Related work

Existing work on specification animation either supports the process of model checking or executing formal specifications by means of translating them into executable code. To the best of our knowledge, no related work supports the same technique proposed in this paper.

One of the early studies on requirements specification animation was done by Kramer and Ng in Kramer and Ng (1988). They built an animator to provide facilities for selecting and executing transactions in order to show the behavior of a particular scenario specified in the requirements specification. Specification animation actions include graphically depicting input–output mappings, controlling the triggering of actions, replaying and interacting with transactions. Oliver and Kent explored the technique for animating formal specifications written in Object Constraint Language (OCL) for UML (Oliver and Kent, 1999). The essential idea is to first generate execution paths from the post-condition of an operation, and then evaluate each term on the path, and finally apply invariants to the after-states to produce a set of "solutions". Gogolla et al. developed an approach for the validation of UML models and OCL constraints based on animation and certification and a software tool, known as USE tool (UML-based Specification Environment), to support the execution of UML models and the checking of OCL constraints (Gogolla et al., 2007). Boanti et al. construct a graphical animator known as AsmetaA to support animation of Abstract State Machines (Bonfanti et al., 2018). The animator can demonstrate the execution of state-based specifications by means of showing a sequence of states using graphical elements, such as tables and colors. The common characteristic of these studies is that the animation of specifications are based on some kind of "action sequence" derived from the specification.

B-Toolkit is an early tool to support both construction and analysis of specifications in B-Method through animation, testing and proof (Bicarregui et al., 1997). ProB for B-Method is another toolset that checks the consistency of B specifications via model checking and also graphically displays the path of state transitions in a counter-example when a violation of the invariant concerned is discovered (Leuschel and Butler, 2003, 2008). ProB also supports step-by-step animation of B-machines that may involve non-deterministic operations (Leuschel et al., 2014). Specifically, it provides the user with a description of the current state of the machine, the history of the state changes up to the current state, and a list of all the enabled operations along with proper argument instantiations. There are also other

tools offering the facilities that can be used for animation of specifications in the B-Method, such as CLPS-B (Bouquet et al., 2002) and the BZ-Testing-Tools (Ambert et al.). ProB is also extended to support refinement animation for Event-B (Hallerstede et al., 2010) to detect a variety of errors that occur frequently during refinement. Mashkoor describes a software tool called Brama to support the animation of Event-B specifications for their validation (Mashkoor, 2011). The animation aims to demonstrate the selected behavioral scenarios each of which represents a sequence of events. The animation of a model can be seen through the variable-view, machine-view, and event-view. The most interesting is the machine graph-view, which shows the animation of all the refinements of the model in one glance. All the refinement levels are animated concurrently. During the animation, Brama finds out whether the related invariant clause has been violated and indicates the violated part of the invariant if the violation occurs. Further development along this line has been made by the Mashkoor and his colleagues to provide a behavior-preserving transformation approach to dealing with the animation of non-animatable formal specifications in Event-B (Mashkoor and Jacquot, 2017; Mashkoor et al., 2017). Non-animatable specifications are heuristically transformed into equivalent animatable specifications for an appropriate animation. Specification animation is used as an auxiliary means to support valid refinement steps as well. Jacquot and Mashkoor discuss the issue of the validation of formal specifications and the related techniques including specification animation (Jacquot and Mashkoor, 2018). Alloy is a formal language for describing structural properties and its tool supports two kinds of analysis: simulation and checking (Jackson, 2002). The simulation is designed to verify the consistency of an invariant or to demonstrate an operation by generating a state or transition. The checking is aimed at testing the consequence of the specification by attempting to generate a counterexample. Another tool that adopts model checking for presenting the dynamic behavior of systems is UPPAAL (Behrmann et al., 2004; Vaandrager, 2011). In this system, the user can 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 reported an approach to animating formal specifications in Parnas' SCR tabular notation (Gargantini and Riccobene, 2003). 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 (Fitzgerald et al., 2008), and has been used in some industrial projects (Larsen and Fitzgerald, 2007; Kurita et al., 2008b). 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 (Larsen et al., 2010a) provides functions similar to VDMTools, and is built on an open, extensible platform based on the Eclipse framework. Using the combinatorial testing technique (Larsen et al., 2010), 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) (Owre et al., 1992) 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 (Shankar, 1999). And PVS specifications can be animated by displaying the results of the ground evaluator in the Graphical User Interfaces (GUIs) created by Tcl/Tk (Crow et al., 2001). These animation techniques can be characterized by using formal proof or model checking to verify relevant invariants or properties during the animation process.

There are also studies on specification animation based on Z notation or other specification languages. PiZA (Hewitt et al., 1997) 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 (Morrey et al., 1998). 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 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 Najafi and Haghighi (2011), 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 (Miller and Strooper, 2003a). The framework provides a testgraph editor for the user to edit testgraphs, and then derive sequences for animation by traversing the testgraph. Stepien and Logrippo built a toolset to translate LOTOS traces to MSC, and provide a graphic animator (Stepien and Logrippo, 2002). 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 Combes et al. (2002). 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. Moreover, our work is also related to studies on formal specification-based testing by sharing some fundamental ideas on test case generation from specifications or models. For example, Mandrioli et al. describe a method and tool for generating test cases from formal specifications written in the extended temporal-logic-based language known as TRIO that is suitable for writing time-related specifications (or properties) for real-time reactive systems (Mandrioli et al., 1995). This work shares the similar idea with our method for generating test cases in the sense that a test case, including both input value and output value, can be generated from a given formal specification, but since TRIO uses temporal logical formulas with simple data types in specifications, the specific algorithms for generating test cases from various logical formulas and their complexity differ considerably from those in our work. Gery et al. also present a UML model-based testing approach in their work on a UML-based software development tool, known as Rhapsody, to support complete model-based iterative life-cycle (Gery et al., 2002). The testing approach features the following steps: (1) specifying tests using scenarios represented by extended sequence diagrams in the model, (2) Rhapsody driving and monitoring the model execution for running tests, (3) reviewing results and pinpoint failures to indicate where the scenario is violated, and (4) fixing the defect and verify by rerunning the test. This approach shares the essential idea of generating tests from specifications in the model with our work, but seems to differ from ours in terms of the way to generate test cases, represent test cases, and run test cases.

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 approach will inevitably limit the capability of animation because as Hayes and Jones' study in their publication Hayes and Jones (1989) shows that specifications using pre- and post-conditions are not (necessarily) executable. By contrast, our animation approach described in this paper does not require any

translation of the specification into code; it can directly perform animation by evaluating the pre- and post-conditions of the processes involved in the target system scenarios for the selected animation cases and expected results. This approach is much easier to implement technically, and is capable of dealing with all pre-post style specifications. The ProB animator seems to have the similar capability for operation animation as our approach, but does not offer facilities for data animation in the same style as ours. To the best of our knowledge, no existing work offers support for data animation with voice explanation and animation of logical expressions or operators used in them, while our work presented in this paper covers these two functions in a comprehensible fashion. OVADO is a tool to support the validation of critical data through the formal verification of the required properties on data to ensure that the data comply with their requirements (Fredj et al., 2017). The suggested approach in this tool seems to share the same principle of modeling first and validation second, but it differs from our work in the sense that formal properties on data need to be formed and formal verification of them are required while no formal properties need to be formed and animation data are required in our work.

Our work presented in this paper has also made some contributions different from our own previous publications related to specification animation. In the publication (Liu and Wang, 2007), we describe a technique for animation of a system functional scenario in which all of the operations are defined using an explicit, executable specification. The animation is implemented by first automatically translating it into a Message Sequence Chart (MSC) and then executing the MSC. Different from this work, we deal with the animation of individual operations that are defined using an implicit specification (i.e., pre- and post-conditions) in this paper. The techniques proposed in the two studies are considerably different, although they share the same purpose to validate specifications. The article in Liu and Nakajima (2010) presents a decompositional way to generate test cases from SOFL formal specifications for testing programs, but this paper focuses on how to validate the specification itself. Although the principle of generating animation data may share with that of generating test cases for program testing, the whole contributions are different. The paper (Li and Liu, 2012) first presents our initial idea of how to carry out an animation for a system functional scenario in which all of the operations are specified using pre- and post-conditions. The focus of this work is on the visualized demonstration of the operation "execution" along the system functional scenario based on data flows. Unlike the work, this paper focuses on the animation of each individual operation involved in some system functional scenario. Although showing the transformation from the input of an operation into its output is reflected in both studies, the work introduced in Li and Liu (2012) is much simpler than the one presented in this paper in terms of the techniques used for animation and the animation contents. In Li and Liu (2015), we describe a study to use animation of system functional scenarios as a reading technique for inspection of functional scenarios derived from a CDFD (Condition Data Flow Diagram, part of a SOFL specification), but the work on animation in this paper focuses on animation of individual operations and their related data, not the interactions between operations on the system functional scenarios. The work in Li and Liu (2017) focuses on the introduction of a tool to support the animation of SOFL formal specifications for internal consistency and the translation from the textual expression of the formal specification into a tabular form. Different from the contribution, this paper focuses on the animation of each individual operation, involved data items and logical expressions for validation.

Despite the strengths of our approach mentioned above, some challenges remain and may need to be tackled properly when the approach is actually applied in practice. One is that our approach requires that both pre- and post-conditions of the specification be in a disjunctive normal form (DNF). This restriction may be inconvenient for those who wish to write specifications in a more free style. Although any non-quantified pre- and post-conditions can be automatically transformed into a DNF using a well established algorithm (Rusnak, 2017), such a transformation in some cases (e.g., $(A_1 \wedge B_1) \vee (A_2 \wedge B_2) \vee \cdots \vee (A_n \wedge B_n)$) can lead to an exponential explosion of the formula (e.g., having 2^n terms). Our experience so far suggests that such extreme cases do not happen in SOFL specifications for practical systems, but the resultant specification of the transformation may appear quite differently from the original one in structure and therefore might add some difficulty to the user in understanding the animated behaviors. A way to deal with this problem is that the specification is written directly in DNF from the beginning. Our experience with industry shows that practitioners can easily accept this way of writing formal specifications (Liu et al., 1998b; Luo et al., 2016), since each functional scenario formed from such a specification describes a specific "use case" simple enough for the user to understand easily and the specification writer can easily verify whether all of the possible behaviors of the corresponding operation have been covered in definition. Another challenge is that the operation animation can only demonstrate the relationships between input and output values (of various types) and the data animation can only show the characteristics of the input and output values, but they may not explain comprehensibly what the input–output relationships and the data characteristics actually mean in the real world, which may limit the effect of validation.

8. Conclusion and future work

We have presented a tool-supported specification animation technique that can be used to validate formal specifications by comprehensibly demonstrating three aspects for an operation specified using pre- and post-conditions: (1) the input–output relation of the operation, (2) the characteristics of the relevant data items of the operation with voice explanations, and (3) logical expressions and operators used in the formal specification of the operation. We have discussed how adequate animation data can be generated for performing the animation and described a prototype tool supporting the animation technique. We have also applied the animation technique to an industrial system and compare its effect with the practically well-used specification review technique. Our observation and experience in the application study indicates that the animation technique can bring us four major benefits in validating formal specifications: (1) the user (and the analyst) can easily comprehend the potential behavior of the animated operations, (2) the communication between the user and the analyst can be improved in the early phase of software development before coding, (3) the animation data can be reused as test data for the program to be implemented later, and (4) the validity of the specification can be checked during the animation process.

In the future, we will continue to extend our prototype tool with more features and capabilities for formal specification animation at both operation level and system level. We will also study how animation can be facilitated by appropriate graphical user interface (GUI) and how such a GUI can be automatically produced based on the nature of the related data items of operations. We will also be interested in applying the animation technique with the extended tool to industrial software development projects to find out whether and how the technique can be utilized to support Agile development paradigms in industrial setting.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

We thank our students for their great contributions to the development of the prototype supporting tool for our animation method. We also thank all the anonymous reviewers for their kind efforts in reviewing our manuscript. This work was supported by JSPS KAKENHI Grant Number 26240008 and NSFC of China 61402178.

References

Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Utting, M., Vacelet, N., 2002. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In: Proceedings of FATES 2002. pp. 105–120.

Behrmann, G., David, A., Larsen, K.G., 2004. A tutorial on uppaal. In: Formal Methods for the Design of Real-Time Systems. Springer, pp. 200–236.

Bicarregui, J., Dick, J., Woods, B. Mathews E., 1997. Making the most of formal specification through animation, testing and proof. Sci. Comput. Prog. 29 (1–2), 53–78.

Bonfanti, S., Gargantini, A., Mashkoor, A., 2018. Asmetaa: Animator for abstract state machines. In: Butler, M., Raschke, A., Hoang, T., Reichl, K. (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z. ABZ 2018. In: LNCS 10817, Springer, Southampton, UK, pp. 369–373.

Bouquet, F., Legeard, B., Peureux, F., 2002. CLPS-b- a constraint solver for b. In: Katoen, J.-P.E, Stevens, P. (Eds.), Proceedings of TACAS 2002. In: LNCS 2280, Springer-Verlag, pp. 188–204.

Chen, J., Liu, S., 1996. An approach to testing object-oriented formal specifications. In: Proceedings of the International Conference TOOLS Pacific 96: Technology of Object-Oriented Languages and Systems. TOOLs/ISE, Melbourne.

Combes, P., Dubois, F., Renard, B., 2002. An open animation tool: Application to telecommunication systems. Comput. Netw. Int. J. Comput. Telecommun. Netw. 40 (5), 599–620.

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

Dutertre, B., 2014. Yices 2.2. In: Biere, A., Bloem, R. (Eds.), Computer-Aided Verification (CAV'2014). In: LNCS, Vol. 8559, Springer, pp. 737–744.

Fitzgerald, J., Larsen, P.G., Sahara, S., 2008. Vdmtools: Advances in support for formal modeling in VDM. ACM Sigplan Notices 43 (2), 3.

Fredj, M., Leger, S., Feliachi, A., Ordioni, J., 2017. OVADO: enhancing data valiation for safety-critical railway systems. In: 2017 International Conference on reliability, Safety and Security of Railway Systems (RSSRAIL 2017). IEEE press, Pistoia, Italy, pp. 87–98.

Gargantini, A., Riccobene, E., 2003. Automatic model driven animation of SCR specifications. In: Pezzè, Mauro (Ed.), FASE 2003. In: LNCS, Vol. 2621, Springer, Warsaw, Poland.

Gery, E., Harel, D., Palachi, E>, 2002. Rhapsody: A complete life-cycle model-based development system. In: Proceedings of International Conference on Integrated Formal Methods (IFM 2002). IEEE Press, Turku, Finland, pp. 1–10.

Gogolla, M., Buttner, F., Richters, M., 2007. USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Prog. (69), 27–34.

Hallerstede, S., Leuschel, M., Plagge, D., 2010. Refinement animation for event-b - towards a method of validation. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (Eds.), 2nd International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010). In: LNCS, Vol. 5977, Springer.

Hayes, I., Jones, C.B., 1989. Specifications are not (necessarily) executable. Softw. Eng. J. 4 (6), 330–339.

Hazel, Daniel, Strooper, Paul, Traynor, Owen, 1997. Possum: an animator for the SUM specification language. In: 1997 Asia-Pacific Software Engineering Conference and International Computer Science Conference. IEEE Computer Society Press, pp. 42–51.

Hewitt, M., O'Halloran, C., Sennett, C., 1997. Experiences with piza: an animator for z. In: ZUM'97. In: LNCS, Vol. 1212, Springer, pp. 37–51.

Jackson, D., 2002. Alloy: A lightweight object modeling notation. ACM Trans. Softw. Eng. Methodol. 11 (2), 256–290.

Jacquot, J.P., Mashkoor, A., 2018. The role of validation in refinement-based formal software development. In: Models: Concepts, Theory, Logic, Reasoning and Semantics. College Publications, pp. 202–219.

Knight, J.C., DeJong, C.L., Gibble, M.S., Nakano, L.G., 1997. Why are formal methods not used more widely? In: Holloway, C.M., Hayhurst, K.J. (Eds.), Fourth NASA Langley Formal Methods Workshop. 3356, Hampton, Viginia, pp. 1–12.

Kramer, J., Ng, K., 1988. Animation of requirements specifications. Softw. - Pract. Exp. 18, 749–774.

Kurita, T., Chiba, M., Nakatsugawa, Y., 2008a. Application of a formal specification language in the development of the mobile feliCa IC chip firmware for embedding in mobile phones. In: Cuellar, J., Maibaum, T., Sere, K. (Eds.), Proceedings of FM 2008: Formal Methods. In: LNCS 5014, Springer-Verlag, Turku, Finland, pp. 425–429.

Kurita, T., Chiba, M., Nakatsugawa, Y., 2008b. 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. Springer, pp. 425–429.

Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M., 2010a. The overture initiative integrating tools for VDM. SIGSOFT Softw. Eng. Notes 35 (1), 1–6.

Larsen, P.G., Fitzgerald, J., 2007. Triumphs and challenges for the industrial application of model-oriented formal methods. Tech. Rep. CS-TR-999, University of Newcastle upon Tyne.

Larsen, P.G., Lausdahl, K., Battle, N., 2010. Combinatorial testing for VDM. In: 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2010). pp. 278–285. http://dx.doi.org/10.1109/SEFM.2010.32.

Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D., 2014. From animation to data validation: The prob constraint solver 10 years on. In: Boulanger, Jean-Louis (Ed.), Formal Methods Applied To Complex Systems: Implementation of the B Method. Wiley ISTE, Hoboken, NJ, pp. 427–446.

Leuschel, M., Butler, M., 2003. Prob: A model checker for b. In: Araki, Keijiro, Gnesi, Stefania, Mandrioli, Dino (Eds.), FME 2003: Formal Methods. In: LNCS 2805, Springer-Verlag, pp. 855–874.

Leuschel, M., Butler, M., 2008. Prob: An automated analysis toolset for the b method. Int. J. Softw. Tools Technol. Transf. 10 (2), 185–203.

Li, M., Liu, S., 2012. Automated functional scenarios-based formal specification animation. In: Proceedings of 19th Asia-Pacific Software Engineering Conference (APSEC 2012). IEEE CS Press, pp. 107–115.

Li, M., Liu, S., 2015. Integrated animation-based inspection into formal design specification construction for reliable software systems. IEEE Trans. Reliab. 65 (1).

Li, S., Liu, S., 2017. A software tool to support scenario-based formal specification for error prevention. In: 7th International Workshop on SOFL+MSVL (Submitted). In: LNCS, Springer.

Liu, S., 1999a. 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. In: Lecture Notes in Computer Science, Springer-Verlag, Toulouse, France, pp. 896–914.

Liu, S., 1999b. A Formal Specification and Implementation of a Train Ticket System Using SOFL and Java. Tech. Rep. HCU-IS-99-015, Hiroshima City University.

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

Liu, S., 2018. Agile formal engineering method for software productivity and reliability. In: Proceedings of 2018 Central and Eastern European Software Engineering Conference Russia (CEE-SECR 2018). ACM press, pp. 64–69.

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

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

Liu, S., McDermid, J.A., Chen, Y., 2010. A rigorous method for inspection of model-based formal specifications. IEEE Trans. Reliab. 59 (4), 667–684.

Liu, S., Nakajima, S., 2010. A decompositional approach to automatic test case generation based on formal specifications. In: 4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010). IEEE Press, Singapore, pp. 147–155.

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

Liu, S., Takahashi, K., Hayashi, T., Nakayama, T., 2008. Teaching formal methods in the context of software engineering. In: Proceedings of 1st International Workshop on Formal Methods Education and Training (FMET 2008). NII GRACE Software Engineering Center Technical Report, Kitakyushu City, Japan, pp. 3–12.

Liu, S., Wang, H., 2007. An automated approach to specification animation for validation. J. Syst. Softw. (80), 1271–1285.

Luo, J., Liu, S., Wang, Y., Zhou, T., 2016. Applying SOFL to a railway interlocking system in industry. In: Proceedings of the 6th International Workshop on SOFL+MSVL (SOFL+MSVL 2016). In: LNCS 10189, Springer, Tokyo, Japan, pp. 160–180.
Mandrioli, D., Morasca, S., Morzenti, A., 1995. Generating test cases for real-time systems from logic specifications . ACM Trans. Comput. Syst. 13 (4).
Manthey, N., 2012. Solver Description of RISS 2.0 and PRISS 2.0. KRR Teport 12-02, Knowledge Representation and Reasoning, Technical University Dresden.
Mashkoor, A., 2011. Formal Domain Engineering: From Specification to Validation (Ph.D. thesis). Nancy 2 University, France.
Mashkoor, A., Jacquot, J.P., 2017. Validation of formal specifications through transformation and animation. Requir. Eng. 22 (4), 433–451.
Mashkoor, A., Yang, F., Jacquot, J.P., 2017. Refinement-based validation of event-b specifications. Softw. Syst. Model. 16 (3), 789–8081.
Miller, Tim, Strooper, Paul, 2003a. A framework and tool support for the systematic testing of model-based specifications. ACM TOSEM 12 (4), 409–439.
Miller, Tim, Strooper, Paul, 2003b. A framework and tool support for the systematic testing of model-based specifications. ACM Trans. Softw. Eng. Methodol. 12 (4), 409–439.
Morrey, I., Siddiqi, J., Hibberd, R., Buckberry, G., 1998. A toolset to support the construction and animation of formal specifications. J. Syst. Softw. 41 (3), 147–160.
Moura, L.D., Bjorner, N., 2008. Z3: an efficient SMT solver. In: 11th European Join Conferences on Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08). Springer, pp. 337–340.
Najafi, M., Haghighi, H., 2011. An animation approach to develop c++ code from object-z specifications. In: 2011 CSI International Symposium on Computer Science and Software Engineering (CSSE 2011). IEEE, pp. 9–16.
Oliver, I., Kent, S., 1999. Validation of object oriented models using animation. In: Proceedings of 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium. IEEE Press.
Owre, S., Rushby, J., Shankar, N., 1992. PVS: A prototype verification system. In: Automated Deduction CADE-11. Springer, pp. 748–752.
Parnas, D.L., 2010. Really rethinking formal methods. Computer 43 (1), 28–34.
Rusnak, P., 2017. Tranformation of boolean expression into disjunctive or conjunctive normal form. Cent. Eur. Res. J. 3 (1), 43–49.
Shankar, N., 1999. Efficiently executing PVS. Tech. Rep., Project report, ComputerScience Laboratory, SRI International, Menlo Park.
Stepien, B., Logrippo, L., 2002. Graphic visualization and animation of LOTOS execution traces. Comput. Netw. Int. J. Comput. Telecommun. Netw. 40 (5), 665–681.
Tedre, M., Moisseinen, N., 2014. Experiments in computing: A survey. Sci. World J. 1–11.
Vaandrager, F., 2011. A first introduction to uppaal. In: Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook, Vol. 18.
Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J., 2009. Formal methods: Practice and experience. ACM Comput. Surv. 41 (4), 1–39.

Shaoying Liu is currently a Professor of Software Engineering at Hiroshima University, Japan. Previously, he was a Professor at Hosei University from April 2000 to March 2020. He received the Ph.D. in Computer Science from the University of Manchester, U.K in 1992. His research interests include Formal Methods and Formal Engineering Methods for Software Development, Specification Verification and Validation, Specification-based Program Inspection, Automatic Specification-based Testing, Testing-Based Formal Verification, and Intelligent Software Engineering Environments. He has published a book entitled "Formal Engineering for Industrial Software Development" with Springer-Verlag, twelve edited conference proceedings, and over 200 academic papers in refereed journals and international conferences. He proposed to use the terminology of "Formal Engineering Methods" in 1997, and has established Formal Engineering Methods as a research area based on his extensive research on the SOFL (Structured Object-Oriented Formal Language) method since 1989, and the development of ICFEM conference series since 1997. In recent years, he has served as the General Chair of ICFEM 2017 and PC member for numerous international conferences. He is an Associate Editor for IEEE Transactions on Reliability and the Journal of Software Testing, Verification and Reliability (STVR), respectively. He is IEEE Fellow, BCS Fellow and member of JSSST.

Weikai Miao is an Associate Professor at East China Normal University. He received the Ph.D. in Information Science at Hosei University, Japan in 2014. His research interests include Formal Engineering Methods, SOFL, Software Testing, and Software Verification and have made over 50 publications in various refereed journals and international conferences.