Automatic Test Case and Test Oracle Generation Based on Functional Scenarios in Formal Specifications for Conformance Testing

Shaoying Liu, *Fellow, IEEE* and Shin Nakajima

Abstract—Testing a program to confirm whether it consistently implements its requirements specification is a necessary but time-consuming activity in software development. Automatic testing based on specifications can significantly alleviate the workload and cost, but faces a challenge of how to ensure that both the user's concerns in the specification and possible execution paths in the program are all covered. In this paper, we describe a new method, called "Vibration-Method" or simply "V-Method" , for automatic generation of test cases and test oracle from model-based formal specifications, aiming to address this challenge. The proposed method is suitable for testing information systems in which rich data types are used. Supporting the principle of "divide and conquer" , the method provides a specific technique for generating test cases based on functional scenarios defined in the specification, test case generation criteria, automatic test case generation algorithms, and a well-defined mechanism for deriving test oracle. We elaborate on the method by discussing how initial test cases can be automatically generated, how additional necessary test cases are produced using the "vibration" technique, and how a test oracle can be automatically derived for a group of test cases. We also describe a controlled experiment to evaluate the effectiveness of the method and discuss the important issues in relation to the performance and applicability of the method.

Index Terms—Specification-based testing, black-box testing, functional testing, model-based testing, automatic testing

1 Introduction

A successful software project must ensure the conformance of the implemented program to the user's (or client's) requirements specification [1]. Since the user generally pays more attention to the behaviors of the program than its internal structure, specification-based testing is a suitable technique for checking the conformance [2], [3], [4]. Although program correctness proof [5], [6] and software model checking [7], [8] can be used for a similar purpose and their application can be well justified especially for safety-critical systems, they seem difficult to replace testing to become the main stream of the technologies for verification and validation in practice due to many limitations and constraints [9], [10]. As a required function is usually implemented by a single or a set of program paths in the program [11], conformance testing should ideally cover both the required functions in the specification and all of the relevant program paths in the program. This, however, poses a challenge.

In principle, specification-based testing requires test cases to be generated only based on the specification, without taking the program structure into account. This may result in a situation where all of the required functions are checked but some program paths are still untested. One solution to this problem, which is commonly practiced, is to carry out a structural testing in which test cases can be deliberately generated to meet the path coverage [12], [13]. According to studies in the literature [14] and our own experiences in collaboration with industry (e.g., The Nippon Signal Co., Ltd. in Japan), practitioners are extremely interested in full automation of such a conformance testing since it can significantly help us save budget, time, and avoid human mistakes. In the meanwhile, the automation is also expected to facilitate humans (e.g., the user, tester, developer) to easily confirm whether every distinct, required function is tested properly, because it is humans who have to take the ultimate responsibility for the quality of the program. However, as is well recognized by the research community [15], automatic structural testing is usually difficult, especially for the code with complex and recursive data and program structures. Symbolic execution provides a limited solution for simple and small scale programs but faces serious obstacles for complex and large scale programs [16]. Even if these techniques are feasible in automatic test case generation for some programs, they are always incompetent for automatic derivation of test oracle, which is an indispensable part of testing.

Another solution is to formalize the user's requirements specification to the level at which every required function is defined precisely using a formal notation [17]. Thus, not only can test cases be automatically generated [18], but a well-defined test oracle for test result analysis can be automatically derived [19]. As formal specification techniques have been used for requirements analysis or system design

- *Shaoying Liu is with the Graduate School of Advanced Science and Engineering and School of Informatics and Data Science, Hiroshima University, Higashi Hiroshima 739-8527, Japan. E-mail: sliu@hiroshima-u.ac.jp.*
- *Shin Nakajima is with the Information and Society Research, National Institute of Informatics, Tokyo 101-8430, Japan. E-mail: nkjm@nii.ac.jp.*

Manuscript received 10 Sept. 2019; revised 19 May 2020; accepted 30 May 2020. Date of publication 4 June 2020; date of current version 14 Feb. 2022. (Corresponding author: Shaoying Liu.)
Recommended for acceptance by T. Xie.
Digital Object Identifier no. 10.1109/TSE.2020.2999884

by many companies according to the survey by Woodcock *et al.* [20], research advancement in the area of automatic specification-based testing will demonstrate additional values of formal specification in software development. Even for software projects without using formal specification, this approach can still be flexibly applied, as detailed in Section 6. An interesting but open problem, however, is how test cases can be automatically generated only based on the specification to allow exercise of both all the required functions in the specification and all the relevant program paths in the corresponding program.

In this paper, we put forward a *functional scenario-based vibration test case generation method* as a solution for this problem. We call the method *Vibration-Method* or simply *V-Method* in order to reflect its main characteristic of "vibration" in test case generation. This method is suitable for testing information systems in which rich data types are used. The underlying principle of the V-Method is to convert the formal specification into an equivalent disjunction of *functional scenarios* and then generate adequate test cases from each functional scenario in a "vibration" manner. Both the test case generation and the "vibration" arrangement are integral parts of our V-Method and they two work together as a whole determines the effectiveness of the V-Method. A functional scenario of an operation defines a specific relation between its input and output. In the context of a formal specification, a functional scenario is usually expressed as a predicate expression, formed by taking both the pre- and post-conditions into account, and expected to define a required function in the user's requirements specification (usually written in natural language). As explained in Section 2.1, all functional scenarios of an operation can be automatically derived using an algorithm we have proposed in our previous work [11], [21]. The overall test strategy advocated by our method is that the generated test cases should cover all of the functional scenarios in the specification and meanwhile aim to cover all of the *representative program paths* (REP) in the program where the REP is a subset of all the possible executable program paths that is derived by unfolding loops as conditional statements. As discussed in detail later, the method presents a theoretical improvement of the commonly used *disjunctive normal form* strategy in [2], and is applicable to any operation specified in terms of pre- and post-conditions.

Specifically, we have made the following contributions in this paper. First, two new criteria for test case generation on the basis of functional scenarios are proposed. One requires that adequate test cases be generated to cover both the defined functional scenarios in the specification and the corresponding representative paths in the program, which differs from existing specification-based testing approaches reported in the literature [2], [22], [23], the other requires the test cases to cover more details of each functional scenario. These two criteria can be realized using both the *input-driven test generation strategy* and the *output-driven test generation strategy*. In the case of the input-driven strategy, values for input variables (i.e., test data) are first selected and then the expected values for output variables (i.e., the expected result) are derived from the specification. In the case of the output-driven strategy, the desirable values for output variables are first selected and then the corresponding test data for input variables will be generated for the chosen expected result. The idea of taking the diversity of outputs into account in test case generation has been proposed by Matinnejad *et al.* in [24], [25], but in the existing work there seems to lack a specific and well-defined way to derive test cases from a given output value. Our method has made a further improvement by providing a specific and well-defined way based on the formal specification. Second, a set of algorithms for automatic test case generation is described. A small part of the algorithms were reported in our previous conference publication [23], but many parts, such as algorithms dealing with sequence type operators, composite type operators, and conjunctions, are newly developed in this work. Third, a "*Vibration*" step, simply called *V-Step*, for heuristically generating test cases with the aim to cover all of the REP is developed. An initial idea of the method and a simple prototype tool was presented at a workshop previously [26], but the idea has been significantly developed into more mature strategy and algorithms for test case generation in this work. Fourth, a mechanism for deriving test oracle for test result analysis is presented on the basis of functional scenarios. Existing similar approaches use the entire pre- and post-conditions to form the test oracle for each test result analysis, but our method uses only the relevant functional scenario, a part of the pre- and post-conditions, for each test result analysis, which is more efficient. Finally, a controlled experiment for evaluating the performance of our method are conducted on a *Universal Card System* developed by our research lab in collaboration with industry.

The remaining part of this paper is organized as follows. Section 2 describes our testing method by focusing the discussion on strategy and criteria for test case generation and the derivation of test oracle. Section 3 presents various algorithms for test case generation from atomic predicates, conjunctions, and disjunctions. Section 4 describes the experiment on evaluating the effectiveness of our method using mutation testing. Section 5 introduces related studies to discuss the state-of-the-art in the field of formal specification-based testing and to indicate the position of our contribution in this paper. Finally, in Section 6 we conclude the paper and in Section 7 we point out future research directions.

2 Test Strategy and Criteria

As mentioned previously, the overall test strategy of our method is to automatically generate test cases from a formal specification to cover all of the functional scenarios in the specification and meanwhile attempt to achieve the path coverage of the program as much as possible. Fig. 1 shows the testing process using the strategy. The formal specification, which results from a formalization of the user's informal requirements specification (i.e., requirements specified in natural language), is taken as the very ground for producing an equivalent *functional scenario form* (FSF) (disjunction of functional scenarios). The FSF is then treated as an alternative representation of the specification and used as the basis for test case generation. As a result, necessary test cases and the corresponding test oracles are made available for carrying out testing.

The result of the testing includes two parts: one is the set of output values and the other is a set of traversed paths.

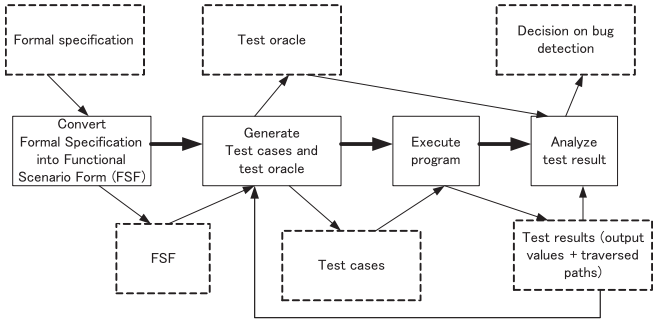


Fig. 1. The testing process.

On the basis of the test oracle and the test result, analysis for determining whether errors are found can then be performed automatically. Also, the information of the traversed paths will be used in decision-making for further testing.

In this method, the concept of *functional scenario* plays a fundamental role and most discussions in this paper depend on it. The concept was first proposed in our ICFEM 2005 article [27] and finalized in our another publication on specification-based program inspection [11], but how it can be used for test case generation was not discussed. For the purpose of this paper, we first introduce the concept in the manner suitable for testing and then describe the overall testing strategy based on it.

2.1 Functional Scenario - Background

Conceptually, the specification of an operation S is denoted as $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$, where S_{iv} is the set of all the input variables whose values are not changed by the operation, S_{ov} is the set of all the output variables whose values are produced or updated by the operation, and S_{pre} and S_{post} are the pre- and post-conditions of S, respectively. For the sake of simplicity, we assume that the related invariant, say inv, on state variables has been incorporated into S_{pre} and S_{post} properly (as conjunction of inv and the original pre- or post-condition), and both the pre- and post-conditions do not contain quantified expressions.

In addition, to write pre- and post-conditions of operations, we adopt the Structured Object-Oriented Formal Language (SOFL) [28], which properly integrates the well-known formal notation VDM-SL [29] and data flow diagrams. Our discussion is not dependent on any specific formal notation, but we need to choose one for expressing the ideas concretely in the discussion. We use SOFL partly because of our expertise and partly because its syntax is comprehensible to readers with experience in programming languages. In the post-condition, we use x and $\overleftarrow{x}$ to represent the value of state (or external) variable x in pre-state and in post-state of the operation, respectively. Thus, $\overleftarrow{x}$ serves as an actual input variable while x is treated as an output variable of the operation; that is, $\overleftarrow{x} \in S_{iv}$ and $x \in S_{ov}$.

Let P be a program implementing S. To ensure that P implements S correctly, the obligation to be discharged is

$$\forall\ \sigma \in \Sigma \cdot S_{pre}(\ \sigma) \Rightarrow S_{post}(\ \sigma, P(\ \sigma)), \tag{1}$$

where Σ denotes all the possible states of S. In this quantified expression, S is perceived as an abstraction of P, which generally defines a relation between the initial state σ before the execution of P and the final state $\sigma\ (= P(\ \sigma))$ after the execution of P. P is treated as a function, mapping the initial state (the pre-state) σ to the final state (the post-state) σ. Condition (1) requires that, for any initial state σ of the program satisfying the pre-condition S_{pre}, the final state σ resulting from the execution of P satisfies the post-condition S_{post}.

Theoretically, the verification of condition (1) using testing requires an *exhaustive testing*, trying every possible initial state in the domain of operation P, but as is well-known, this is usually impossible in practice because of the state explosion problem and time constraints, although it may be used for programs whose scope and scale are small. As briefly explained in the previous section, our method aims to generate adequate test cases from specification S to cover all of the defined functions that reflect the user's distinct functional requirements. Such a function can actually be represented, formally, by a functional scenario defined below.

Definition 1 (functional scenario). *Let* $S_{post} \equiv (G_1 \wedge D_1) \vee (G_2 \wedge D_2) \vee \cdots \vee (G_n \wedge D_n)$, *where each* G_i $(i \in \{1, \ldots, n\})$ *is a predicate called a* guard condition *that contains no output variable in* S_{ov} *and* D_i *a* defining condition *that contains at least one output variable in* S_{ov} *but no guard condition. Then, a* functional scenario *of* S *is defined as a conjunction* $S_{pre} \wedge G_i \wedge D_i$, *and the expression* $(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \cdots \vee (S_{pre} \wedge G_n \wedge D_n)$ *is called a* functional scenario form *(FSF) of* S.

We treat a conjunction $S_{pre} \wedge G_i \wedge D_i$ as a functional scenario (FS) because it defines a distinct function: when $S_{pre} \wedge G_i$ is satisfied by the initial state (or intuitively by the input variables), the final state (or the output variables) is defined by the defining condition D_i. The separation of the defining condition from the conjunction of the pre-condition and the guard condition will facilitate us in test data generation based on the conjunction and in forming test oracle based on both the conjunction and the defining condition. Note that in the pre- and post-conditions of a specification, we treat both a relation (e.g., $x > y$) and its negation as an atomic predicate. Any pre-post specification can be translated into an equivalent FSF using a well-established algorithm whose content and implementation are described in our previous publications [11] and [21], respectively.

Definition 2 (complete specification). *Let* $(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \cdots \vee (S_{pre} \wedge G_n \wedge D_n)$ *be a FSF of operation specification* S. *Then,* S *is said to be complete if and only if the condition* $G_1 \vee G_2 \vee \cdots \vee G_n \Leftrightarrow true$ *holds.*

Note that the definition of completeness of an operation specification here may differ from the conventional understanding that requires the specification to define all of the user's requirements. Rather, our definition requires that any input satisfying the pre-condition must satisfy one of the guard conditions (e.g., G_1), thus guaranteeing that the output of the operation can be defined by the corresponding defining condition (e.g., D_1), provided that all of the defining conditions are satisfiable. As we will discuss in Section 4.5 later, the completeness of a specification impacts the effectiveness of our testing method.

Example 1. Let us take an operation known as $Check\_Triangle$ as an example to explain these basic notions.

This operation takes three sides of a triangle as input and determines the type of the triangle, including *equilateral triangle* , *isosceles triangle*, *other triangle*, and *non-triangle* . Its formal specification written in the SOFL language is given below where the process can be understood as an operation in general sense.

process $Check\_Triangle(d1, d2, d3 : int)$
$t\_type : string$
pre $true$
post $(d1 = d2 \text{ and } d2 = d3 \text{ and}$
$t\_type = \text{" equilateral triangle"}) \text{ or}$
$((d1 = d2 \text{ or } d1 = d3 \text{ or } d2 = d3) \text{ and}$
$t\_type = \text{" isosceles triangle"}) \text{ or}$
$(d1 \neq d2 \text{ and } d1 \neq d3 \text{ and } d2 \neq d3 \text{ and}$
$t\_type = \text{" other triangle"}) \text{ or}$
$((d1 \leq 0 \text{ or } d2 \leq 0 \text{ or } d3 \leq 0) \text{ and}$
$t\_type = \text{" non-triangle"})$
end_process

In the context of the notation $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$, the four parts of the operation $Check\_Triangle$ are as follows:

$Check\_Triangle_{iv} = \{d1, d2, d3\}$
$Check\_Triangle_{ov} = \{t\_type\}$
$Check\_Triangle_{pre} = \ true$
$Check\_Triangle_{post} = \ (d1 = d2 \text{ and } \ldots$ (omitted for brevity).

A functional scenario form can be derived from this specification, which includes the following four functional scenarios:

(1)	G_1 :	$d1 = d2 \text{ and } d2 = d3$
	D_1 :	$t\_type = \text{" equilateral triangle"}$
	FS_1 :	$G_1 \text{ and } D_1$
		$(\Leftrightarrow Check\_Triangle_{pre} \text{ and } G_1 \text{ and } D_1)$
(2)	G_2 :	$(d1 = d2 \text{ or } d1 = d3 \text{ or } d2 = d3)$
	D_2 :	$t\_type = \text{" isosceles triangle"}$
	FS_2 :	$G_2 \text{ and } D_2$
(3)	G_3 :	$d1 \neq d2 \text{ and } d1 \neq d3 \text{ and } d2 \neq d3$
	D_3 :	$t\_type = \text{" other triangle"}$
	FS_3 :	$G_3 \text{ and } D_3$
(4)	G_4 :	$(d1 \leq 0 \text{ or } d2 \leq 0 \text{ or } d3 \leq 0)$
	D_4 :	$t\_type = \text{" non-triangle"}$
	FS_4 :	$G_4 \text{ and } D_4$

2.2 Test Case Generation Criteria

Before discussing criteria for test case generation, we need to clarify the basic concepts, such as *test data*, *test case*, *test set*, *test suite*, *test oracle*, and *test condition* of a functional scenario, because some of them are used in the literature with slightly different meanings.

Definition 3 (test data). *A test data is an assignment of values to all of the input variables from their types.*

Definition 4 (test case). *A test case is a test data together with the expected values for the output variables. Let* $S_{iv} = \{x_1, x_2, \ldots, x_n\}$ *and* $S_{ov} = \{y_1, y_2, \ldots, y_m\}$ *be the set of all input variables and output variables of operation* S*, respectively. Let* $Type(z)$ *denotes the type of input variable or output variable* z*, where* $z \in \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$*. Then, a test case, denoted by* tc*, is a mapping from the union of* S_{iv} *and* S_{ov}*to the set* $Values$*:*

$$tc : S_{iv} \cup S_{ov} \rightarrow Values$$
$$tc(z) \in Type(z),$$

where $Values = Type(x_1) \cup \cdots \cup Type(x_n) \cup Type(y_1) \cup \cdots \cup Type(y_n)$.

A test case is usually expressed as a set of pairs of input variables or output variables with their value. For example, $tc = \{(x_1, 5), \ldots, (x_n, 20), (y_1, 15), \ldots, (y_m, 60)\}$ is a possible test case.

Definition 5 (test set). *A test set is a collection of test cases, and is usually expressed as a set of sets of pairs. A test suite is a test set expected to run for a specific purpose.*

Definition 6 (test condition). *Let* $S_{pre} \wedge G \wedge D$ *be a functional scenario of operation* S*. Then, the conjunction of* $S_{pre} \wedge G$ *is called* test condition *of the functional scenario.*

Since the test condition $S_{pre} \wedge G$ contains only input variables of the operation, test data (or test cases without expected result) for checking whether defining condition D is implemented correctly will be produced only based on the test condition. The defining condition will be used to form a test oracle for test result analysis, as discussed in Section 2.3.

Definition 7 (restricted domain). *The* restricted domain *of operation* S *is the subset of the domain of the operation in which every value satisfies the pre-condition of the operation specification.*

The above notions are concerned with the specification, but we also need some fundamental concepts on programs in the proposed criteria for test case generation below. They are defined next, respectively.

Definition 8 (program). *A program is a five-tuple* (S_0, SN, CN, R, T)*, where* S_0 *denotes the starting node,* SN *the set of statement nodes,* CN *the set of condition nodes,* R *the relation between nodes, and* T *the set of terminating nodes, and they satisfy the following conditions:*

1) $S_0 \in SN \cup CN$
2) $R : (SN \cup CN) \times (SN \cup CN)$
$\forall_{n \in CN} \exists!_{n_1, n_2 \in SN \cup CN} \cdot n_1 \neq n_2 \wedge (n, n_1) \in R \wedge (n, n_2) \in R$
3) $T \subseteq SN$

A program can be represented as a semantically equivalent graph that is composed of the five elements S_0, SN, CN, R, T, and satisfies the above three conditions. Each statement in the program, including *empty statement* (denoted by *skip*), is represented by a node in SN called a *statement node*. Each condition (a predicate) is represented by a node in CN called a *condition node*. The connections between the nodes (including both statement and condition nodes), which reflect the control flows between statements, are represented by relation R. The starting node S_0 denotes the very first statement or condition of the program. A terminating node in T represents a statement that must be executed lastly in an execution of the program. A program is also characterized by the fact that every condition node is connected to two unique other nodes each of which is either a statement node or condition node as defined in condition (2) above.

Definition 9 (program path). *Let* $P = (S_0, SN, CN, R, T)$ *be a program. A program path* p *of* P *is a sequence of nodes* $[S_0, n_1, n_2, \ldots, n_m]$*, where* $n_m \in T$*, and we let* $E(p) = \{(S_0, n_1), (n_1, n_2), \ldots, (n_{m-1}, n_m)\}$.

A program path is a sequence of nodes, beginning with the starting node of the program and ending with one of the terminating nodes. The path can also be viewed as a relation composed of all the edges of the path. To avoid any possible confusion, we use $E(p)$ to represent the corresponding relation of path p.

Definition 10 (representative program paths). *Let $P = (S_0, SN, CN, R, T)$ be a program. Then, we use $Rpp(P)$ to denote a set of representative program paths of P that must satisfy the following two conditions:*

1) $\forall_{e \in R} \exists_{p \in Rpp(P)} \cdot e \in E(p)$
2) $\forall_{p \in Rpp(P)} \cdot \neg \exists_{p1 \in Rpp(P)} \cdot E(p) \subseteq E(p1)$.

$Rpp(P)$ does not represent a unique set of representative program paths of P, according to the definition, but it contains all of the program paths that cover all of the control flow edges of P, as indicated in condition (1), and does not contain different program paths that overlap control flow edges in their corresponding relations, as implied by condition (2).

In this paper, we always use $Rpp(P)$ to denote the set of representative program paths of interest that is determined based on the requirement of testing.

Definition 11 (traversed path). *Let $P = (S_0, SN, CN, R, T)$ be a program and p be a program path of P. When P is executed, if all of the nodes of p are executed (in the case of a statement) or evaluated (in the case of a condition) in the order of their appearance on the path, we say that p is traversed.*

With the definitions of the above concepts, we can now define criteria for test case generation next.

Criterion 1. *Let P be a program implementing operation specification S; $(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \cdots \vee (S_{pre} \wedge G_n \wedge D_n)$ be an FSF of S, where $(n \geq 1)$. Let T be a test set generated from S for testing P. Then, T is expected to satisfy the following four conditions:*

1) $\forall_{i \in \{1,\ldots,n\}} \exists_{tc \in T} \cdot S_{pre}(tc) \wedge G_i(tc)$
2) $\neg(G_1 \vee G_2 \vee \cdots \vee G_n \Leftrightarrow true) \Rightarrow$
   $\exists_{tc \in T} \cdot S_{pre}(tc) \wedge \neg(G_1 \vee G_2 \vee \cdots \vee G_n)(tc)$
3) $\exists_{tc \in T} \cdot \neg(S_{pre}(tc))$
4) $\forall_{p \in Rpp(P)} \exists_{tc \in T} \cdot traversed(p, tc)$.

We call this criterion *scenario-path coverage* (SPC). The criterion suggests that a generated test set T cover all of the functional scenarios in the specification and all of the representative paths in the program. Specifically, the test set T is expected to meet four conditions: (1) for each functional scenario, there must exist a test case in T that satisfies the test condition of the functional scenario, (2) if the disjunction of all the guard conditions does not cover the entire restricted domain of the operation (i.e., the disjunction is not equivalent to $true$), there must exist a test case in T that satisfies the negation of the disjunction of all the guard conditions, (3) there must also exist a test case that violates the precondition, and (4) for each representative program path, there must exist a test case that makes the path traversed (denoted by $traversed(p, tc)$).

This test criterion is consistent with the well known criteria reported in the literature. Conditions (1) and (2) together reflect the equivalence partitioning technique for generating test cases [30], [31] on the basis of precisely defined test conditions of functional scenarios. Condition (3) does not help determine whether an error is found by the test case because the semantics of the pre-post style specification in SOFL allows any behaviors of the implemented program when the pre-condition is violated, which is consistent with the well-established refinement theory [32], but it can play a similar role to that of the robustness testing technique [33] and help the tester or developer judge whether the program's behavior is acceptable for the user in this circumstance. Condition (4) corresponds to simple path coverage testing [31].

Another criterion that requires the generated test set to cover more detailed parts of a test condition allows more detailed situations to be examined in testing.

Criterion 2. *Let test set T satisfy* Criterion 1; *$S_{pre} \wedge G$ be the test condition of the functional scenario $S_{pre} \wedge G \wedge D$ in the FSF of operation S; and $P_1 \vee P_2 \vee \cdots \vee P_m$ be a disjunctive normal form (DNF) of $S_{pre} \wedge G$. Then, T must also satisfy the condition*

$$\forall_{P \in \{P_1, P_2, \ldots, P_m\}} \exists_{tc \in T} \cdot P(tc).$$

This criterion imposes further partition of the sub-domain defined by the test condition of a functional scenario involved in *Criterion 1*. Specifically, it requires that for each constituent clause of the DNF of the test condition, T contains a test case satisfying it. This means that at least one test case from each partitioned area of the sub-domain is required to select for the testing. Assuming that clause P is the conjunction $Q_1 \wedge Q_2 \wedge \cdots \wedge Q_w$, where Q_j $(j = 1, \ldots, w)$ is an atomic predicate or its negation, the predicate $P(tc)$ will be equivalent to $Q_1(tc) \wedge Q_2(tc) \wedge \cdots \wedge Q_w(tc)$. Since any predicate expression can be converted into an equivalent DNF, this criterion is applicable to any test condition of any functional scenario.

Similarly to existing criteria for testing case generation (e.g., branch coverage, path coverage), the two criteria above show a guideline for generating adequate test cases, that is, they tell when test case generation should stop. Due to many different reasons, however, these criteria may not always be guaranteed to satisfy in practice (e.g., infeasible paths, time constraints).

Example 2. Let us take an operation called *Purchase_Ticket_From_Card*, *PTFC* for short, as an example to illustrate the meaning of *Criterion 1*. *PTFC* is a sub-operation of the high level operation *PurchaseTicket* in the *Universal Card System* used in our experiment that is described in Section 4. Its formal specification fits for our purpose, and is given below in SOFL.

process $PTFC(status : string,\ fare : nat0)$
$actual\_fare : int$
ext wr $card$: $Card$
pre $fare * 0.5 <= card.buffer$
post $case\ status\ of$
"$Infant$" $->$ $actual\_fare = 0\ and\ card =\ card$;
"$Student$" $->$ $actual\_fare = fare - fare * 0.5\ and$
$card =$
$modify(\ card, buffer ->\ \ card.buffer - actual\_fare)$;
"$Normal$" $->$ $actual\_fare = fare\ and$

TABLE 1
Example of Test Set Satisfying Criterion 1

No. of test cases	Input variables			Output variables (expected results)	Covered functional scenarios or negation of pre-condition and paths
	$fare$	$status$	$\tilde{}card$		
(1)	380	"$Infant$"	$\tilde{}card.buffer = 1500$	$actual\_fare = 0, card.buffer = 1500$	Scenario (1) and p_1
(2)	1200	"$Student$"	$\tilde{}card.buffer = 2300$	$actual\_fare = 600, card.buffer = 1700$	Scenario (2) and p_2
(3)	530	"$Normal$"	$\tilde{}card.buffer = 3800$	$actual\_fare = 530, card.buffer = 3270$	Scenario (3) and p_2
(4)	960	"$Pensioner$"	$\tilde{}card.buffer = 4300$	$actual\_fare = 672, card.buffer = 3128$	Scenario (4) and p_2
(5)	130	"$Disable$"	$\tilde{}card.buffer = 4100$	$actual\_fare = 91, card.buffer = 4009$	Scenario (5) and p_2
(6)	240	"$Superman$"	$\tilde{}card.buffer = 5205$	$actual\_fare = -1, card = \tilde{}card$	Scenario (6) and p_3
(7)	1500	"$Anything$"	$\tilde{}card.buffer = 1200$	$actual\_fare = -1, card = \tilde{}card$	negation of pre-condition and p_3

$card =$
$modify(\ card, buffer -> \ \ card.buffer - actual\_fare);$
"$Pensioner$" $->\ \ actual\_fare = fare - fare * 0.3$ *and*
$card =$
$modify(\ card, buffer -> \ \ card.buffer - actual\_fare);$
"$Disable$" $->\ \ actual\_fare = fare - fare * 0.3$ *and*
$card =$
$modify(\ card, buffer -> \ \ card.buffer - actual\_fare);$
$default -> \ \ actual\_fare = -1$ *and* $card = \ card$
end_process

where $nat0$ is the type of natural numbers including zero (i.e., $\{0, 1, 2, 3, \ldots\}$), and the composite type $Card$ is declared as follows:

$Card\ =$ *composed of*
id: *string*
buffer: $nat0$
...
end;

A card contains several fields, including an id, a *buffer* containing the current amount of money available for use, and other fields, which are omitted for brevity without affecting the understanding. The specification states that if half of the input $fare$ is not greater than the *buffer* of the $card$, when the input $status$ is one of the five situations "$Infant$", "$Student$", "$Normal$", "$Pensioner$", and "$Disable$", the output $actual\_fare$ is properly calculated and the $card$ is properly updated by reducing the $actual\_fare$ from its *buffer* (expressed by the $modify$ operator) whenever applicable; otherwise, the $actual\_fare$ will be given a special value -1, which is denoted by $actual\_fare = -1$, indicating that no ticket can be purchased, and the $card$ remains unchanged. The four parts of the operation $PTFC$ are as follows:

$PTFC_{iv} = \{status, fare,\ card\}$
$PTFC_{ov} = \{actual\_fare, card\}$
$PTFC_{pre} =\ \ fare * 0.5\ <=\ \ card.buffer$
$PTFC_{post} =\ \ case\ status\ of\ \ \ldots$ (omitted for brevity).

Applying the algorithm reported in our previous work [11], a functional scenario form can be derived from this specification, which includes the following six functional scenarios:

1) $fare * 0.5\ <=\ \ card.buffer$ *and* $status =$ "$Infant$" *and* $actual\_fare = 0$ *and* $card = \ card$
2) $fare * 0.5\ <=\ \ card.buffer$ *and* $status =$ "$Student$" *and* $actual\_fare = fare - fare * 0.5$ *and* $card =$ $modify(\ card, buffer -> \ card.buffer - actual\_fare)$
3) $fare * 0.5\ <=\ \ card.buffer$ *and* $status =$ "$Normal$" *and* $actual\_fare = fare$ *and* $card =$ $modify(\ card, buffer -> \ card.buffer - actual\_fare)$
4) $fare * 0.5\ <=\ \ card.buffer$ *and* $status =$ "$Pensioner$" *and* $actual\_fare = fare - fare * 0.3$ *and* $card =$ $modify(\ card, buffer -> \ card.buffer - actual\_fare)$
5) $fare * 0.5\ <=\ \ card.buffer$ *and* $status =$ "$Disable$" *and* $actual\_fare = fare - fare * 0.3$ *and* $card =$ $modify(\ card, buffer -> \ card.buffer - actual\_fare)$
6) $fare * 0.5\ <=\ \ card.buffer$ *and* $status\ notin$ {"$Infant$","$Student$","$Normal$", "$Pensioner$","$Disable$"} *and* $actual\_fare = -1$ *and* $card = \ card$

In scenario (1), for example, $fare * 0.5\ <=\ \ card.buffer$ is $PTFC_{pre}$, $status =$ "$Infant$" the guard condition, and $actual\_fare = 0$ *and* $card = \ card$ the defining condition, to define the two output variables $actual\_fare$ and $card$. The rest of the functional scenarios can be interpreted similarly.

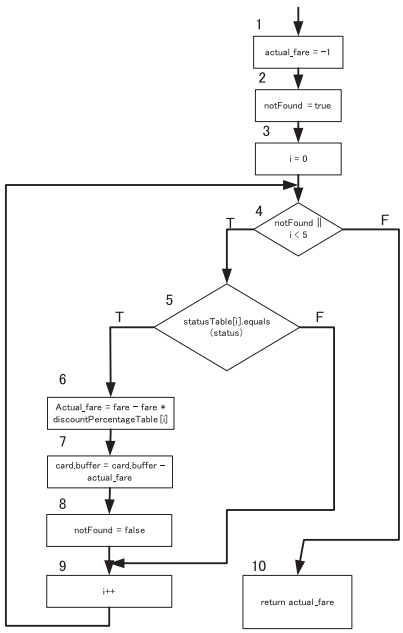
Applying *Criterion 1* to the six functional scenarios above, we generate a test set containing seven test cases that satisfy the six functional scenarios and the negation of the pre-condition, respectively, as shown in Table 1. This test set is generated by first selecting test data for the input variables $fare$, $status$, and $card$ based only on the test condition of each functional scenario, and then derive the corresponding expected values for the output variables $actual\_fare$ and $card$ based on the defining condition of the functional scenario.

Let operation $PTFC$ be implemented as a Java method in a class called $PurchaseTicket$. The five status are stored in an array called statusTable and the corresponding discount percentages (e.g., 0.3 or 30 percent) are stored in another array called discountPercentageTable. The method searches for the input status in statusTable, and once it is found, the corresponding discount percentage in discount PercentageTable will be applied properly in calculating the actual$_f$are. The details of the Java code is given in Fig. 2a and its flowchart is depicted in Fig. 2b for readability. In the flowchart, each statement or decision is attached with a number, thus program paths can be formed in terms of the step numbers for the convenience of discussions in this paper. The program contains three REPs named p_1, p_2, p_3, and another path p_4: $p_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 4, 10]$, $p_2 = [1, 2, 3, 4, 5, 9, 4, 5, 6, 7, 8, 9, 4, 10]$, $p_3 = [1, 2, 3, 4, 5, 9, 4, 10]$, and

```
class PurchaseTicket {
 Card card;
 String statusTable [5];
 double discountPercentageTable [5];
 ...
 public int PTFC (String status, int fare) {
  int actual_fare = -1;
  boolean notFound = true;

  for (int i = 0; notFound || i < 5; i++) {
   if (statusTable[i].equals(status) {
    actual_fare = fare - fare * discountPercentageTable [i];
    card.buffer = card.buffer - actual_fare;
    notFound = false;
     }
    }
    return actual_fare;
}
```

(a)



(b)

Fig. 2. A program intended to implement the specification of operation PTFC.

$p_4 = [1, 2, 3, 4, 10]$. Here p_4 is not executable because the decision in step 4 can never be false in the first evaluation.

The test set in Table 1 satisfies *Criterion 1*, because the test cases cover both all of the functional scenarios and all of the three executable representative program paths. In particular, it is worth mentioning that test case (7) helps find a bug contained in the decision of step 4, notFound $\|$i<5, because it causes the "out of array boundary" error when using test case (7). The correct decision should be notFound $\&$ i$<$5.

2.3 Test Result Analysis

Test result analysis aims to determine whether a test has found any errors or not. It needs not only the test case and the corresponding test result, but also a test oracle. In our V-Method, a test oracle is a condition specific to a functional scenario that can tell whether a test case generated from the scenario finds errors or not. In general, one test oracle is derived from one functional scenario and test oracles derived from different functional scenarios are usually different.

Definition 12 (test oracle). *Let $f \equiv S_{pre} \wedge G \wedge D$ be a functional scenario of operation specification S and P be a program implementing S . Let t be a test case generated from f and r be the result of executing program P using t. Then, an error in P is found using t if the following condition holds:*

$$S_{pre}(t) \wedge G(t) \Rightarrow \neg D(t, r).$$

This condition states that for a test case t satisfying the test condition $S_{pre} \wedge G$, if the execution result r of P does not satisfy the defining condition D (together with t), including the situation where $D(t, r)$ is undefined, it implies that P contains an error, since a correct implementation of the functional scenario must ensure that for any input satisfying the test condition, the defining condition will be met by the output of the program. If specification S has n functional scenarios, n (specific) test oracles can be derived.

Let us take the operation $PTFC$ given in Section 2.2 for example. For brevity, we concentrate only on functional scenario (6) to explain how the test oracle can be used for test result analysis. Applying *Definition* 12, we can form the following test oracle for tests based on the functional scenario:

$$\begin{aligned} &fare * 0.5 <= \ card.buffer \wedge status \ notin \\ &\{''Infant'',''Student'',''Normal'',''Pensioner'',''Disable''\} \\ &\Rightarrow \neg(actual\_fare = -1 \ and \ card = \ card). \end{aligned}$$

Test case (6) in Table 1 is generated based on functional scenario (6). When it is used to run the program in Fig. 2, an "out of array boundary" error occurs, which implies that the output variable $actual\_fare$ and the external variable $card$ are both undefined. This makes the defining condition $(actual\_fare = -1 \ and \ card = \ card)$ undefined and ultimately makes the implication evaluate to undefined as well, according to the extended three-value logic adopted in VDM and SOFL. The undefined situation is treated as a special "value", written as nil, and different from $true$. According to *Definition* 12, the test oracle is not satisfied and therefore the test has found an error in the program, which is in the decision notFound $\|$i $<$ 5, as discussed in Section 2.2.

The advantages of our approach to defining the test oracle are three-fold. First, a test oracle can be automatically derived from one functional scenario and can be automatically analyzed for a definitive decision on errors. Second, the analysis of the test oracle involves only one functional scenario rather than the whole post-condition, which is likely to enhance the efficiency of the analysis system significantly, as compared with the other specification-based approaches that usually involve the whole post-condition for evaluating the test oracle. Finally, the analysis of each test oracle allows the tester to concentrate only on one functional scenario that usually corresponds to a single detailed use case of the system, therefore is helpful for the tester to explain the meaning of each test to the user (or client) in practice.

3 Test Case Generation Algorithms

The criteria described previously define the conditions for generating adequate test cases based on predicate expressions, but how the test case can actually be produced still remains unaddressed.

In this section, we take a bottom-up approach to present algorithms for automatically generating test sets from various levels of predicate expressions, starting from atomic predicates, through conjunctions, and finally completing with the discussion on disjunctions. A few algorithms for test case generation based on numeric types (e.g., integers and real numbers) have been used and tested in a prototype tool reported in our previous work [34], but most of the

TABLE 2
Algorithms for Case (a)

No. of algorithms	$\ominus$	Algorithms of test case generation for x_1	Algorithms of test case generation for the remaining variables ($i = 2,..., q$)
(1)	$=$	$x_1 := E$	$x_i := any \in Type(x_i)$
(2)	$>=$, $<>$ or $>$	$x_1 := E + \sigma$	$x_i := any \in Type(x_i)$
(3)	$<=$ or $<$	$x_1 := E - \sigma$	$x_i := any \in Type(x_i)$

other algorithms introduced in this section are implemented in our latest prototype tool that is briefly introduced in Section 3.7. For the sake of readability, the algorithms will be presented in an abstract manner, focusing on the elaboration on their essential idea and rationale rather than concrete steps as required in code.

Since various data types are provided in SOFL, as well as in other model-based formal notations, such as VDM-SL [35], Z [36], and Event-B [37], we need algorithms to deal with the test set generation from every type of atomic predicate (e.g., atomic predicate of numeric type, set type, sequence type, composite type, and map type). For the sake of space, we only choose the commonly used numeric type, set type, sequence type, and composite type as examples to show the underlying principle of test case generation. Extension to other types can be done by following the same principle.

Note that when the algorithms are applied to generate test cases, we treat input variables and output variables equally because all of them need values in the test cases generated. Of course, if an expected test result is specifically required, the value of output variables can first be determined, which will convert the entire functional scenario into a predicate formula containing only input variables. Then, the predicate is used to generate values for input variables in the test case.

3.1 Generation From Atomic Predicates

Let $Q(x_1, x_2, \ldots, x_q)$ be an atomic predicate mentioned in *Criterion 2* above. The variables $x_1, x_2, \ldots, x_q$ are input or output variables, which are called *i-o variables*, of the related operation but may be part of all the i-o variables x_1, $x_2, \ldots, x_n$ where $n \geq q$. The goal of test case generation from the predicate is to choose such a value for each variable that will satisfy the predicate. To fulfill this goal efficiently, the algorithms for test case generation to be discussed in this section must take the structure of the predicate into account. Since the algorithms vary depending on the number of the variables involved and the form of the predicate, we divide the discussion into the following three situations:

(a) Only one i-o variable (i.e., $q = 1$) is involved and $Q(x_1)$ has the format $x_1 \ominus E$, where $\ominus \in \{=, >, <, >=, <=, <>\}$ is a relational operator and E a constant. The operator $>=$ means "greater than or equal to" , $<=$ means "less than or equal to" ; $<>$ means inequality; and the others are commonly used operators.

(b) Only one i-o variable is involved and $Q(x_1)$ has the format $E_1 \ominus E_2$, where E_1 and E_2 are both arithmetic expressions that involve x_1.

(c) More than one i-o variable is involved and $Q(x_1, x_2, \ldots, x_q)$ has the format $E_1 \ominus E_2$, where E_1 and E_2 are both arithmetic expressions possibly involving all the variables x_1,$x_2, \ldots, x_q$.

Generating a test case to satisfy $x_1 \ominus E$ in case (a) is rather simple, but generating a test case to satisfy $E_1 \ominus E_2$ in case (b) becomes a little more complicated: it needs first to transform the format $E_1 \ominus E_2$ to the format $x_1 \ominus E$, and then to apply the algorithms for case (a). Generating a test case to satisfy $E_1 \ominus E_2$ in case (c) requires more actions:

1) Randomly assigning values from appropriate types to the i-o variables $x_2, \ldots, x_m$ to transform the format into the format $E_1 \ominus E_2$ in case (b).
2) Applying the algorithms for case (b) to the derived format $E_1 \ominus E_2$.

The algorithms for generating test cases from a predicate in case (a) are fundamental and essential because the algorithms dealing with cases (b) and (c) are all dependent on them in the manner mentioned previously. Therefore, we only discuss the details of the algorithms dealing with predicates in case (a) below.

Algorithm 1. *Let $q = 1$ and $Q(x_1, x_2, \ldots, x_q) \equiv x_1 \ominus E$, where $\ominus \in \{=, >, <, >=, <=, <>\}$ and E is a constant. Then, a set of algorithms for generating test cases to satisfy $Q(x_1, x_2, \ldots, x_q)$ is given in Table 2, where σ is a positive integer, which can be produced randomly, and ":=" denotes the assignment operator. Note that each of these algorithms can be used independently.*

According to Table 2, when operator $\ominus$ is equality symbol "=" , a test data for variable x_1 can be produced by the assignment $x_1 := E$, meaning that the result of E is assigned to x_1, whilst the remaining input variables $x_2, x_3, \ldots, x_q$ are assigned any value from their type. When operator $\ominus$ is "$>=$" , "$<>$" or "$>$ " , a test data for variable x_1 can be produced by the assignment $x_1 := E + \sigma$, whilst the remaining input variables $x_2, x_3, \ldots, x_q$ are assigned any value from their type, where σ denotes 1 or any positive integer. When operator $\ominus$ is "$<=$" or "$<$" , a test data for variable x_1 can be generated by the assignment $x_1 := E - \sigma$, whilst the remaining input variables $x_2, x_3, \ldots, x_q$ are assigned any value from their type. Any of these algorithms ensures that a test case generated from $Q(x_1, x_2, \ldots, x_q)$ satisfies predicate Q.

It is worth noticing that the algorithms above are suitable for dealing with expressions of numeric types, but if the expression involves operators on compound types, such as *set type*, we need another group of algorithms for test case generation. These algorithms are summarized in *Algorithm 2* below.

Algorithm 2. *Let $q = 1$ and $Q(x_1, x_2, \ldots, x_q) \equiv E(x_1)$, where x_1 is a variable denoting a single element of a set, or a set of elements, $E(x_1)$ is an expression involving x_1 and an operator defined on set types, such as inset (membership), notin (non-membership), card (cardinality), union (union of sets), inter (intersection of sets),* diff *(difference between sets), subset*

TABLE 3
Algorithms for Test Case Generation From Set Type Expressions

No. of algorithms	Expression $E(x_1)$	Algorithms of test case generation for x_1	Algorithms of test case generation for the remaining variables $(i = 2,...,n)$
(1)	x_1 *inset* E_1	$x_1 := get(E_1)$, E_1 is non-empty	$x_i := any \in Type(x_i)$
(2)	x_1 *notin* E_1	$x_1 := get(Type(x_1) \setminus E_1)$	$x_i := any \in Type(x_i)$
(3)	$card(x_1) = w$	$x_1 := \{a_1, a_2, ..., a_w\}$, where x_1 : *set of* T, and $a_k \in T$, $k = 1, 2, ..., w$	$x_i := any \in Type(x_i)$
(4)	$union(x_1, E_1) = E_2$	$x_1 := E_2 \setminus E_1$	$x_i := any \in Type(x_i)$
(5)	$inter(x_1, E_1) = E_2$	$x_1 := E_2$	$x_i := any \in Type(x_i)$
(6)	$diff(x_1, E_1) = E_2$	$x_1 := E_1$ *union* E_2	$x_i := any \in Type(x_i)$
(7)	$subset(x_1, E_1)$	$x_1 := E_1$	$x_i := any \in Type(x_i)$
(8)	$psubset(x_1, E_1)$	$x_1 := E_1 \setminus \{get(E_1)\}$	$x_i := any \in Type(x_i)$
(9)	$x_1 = E_1$	$x_1 := E1$	$x_i := any \in Type(x_i)$
(10)	$x_1 <> E_1$	$x_1 := \{get(Type(x_1))\} \setminus E_1$	$x_i := any \in Type(x_i)$
(11)	$power(x_1) = E_1$	$x_1 := getLargest(E_1)$	$x_i := any \in Type(x_i)$

(subset), psubset (proper subset), and power (power set). Then, the goal of the algorithm for generating a test case from each kind of atomic predicate is to choose a set value that satisfies the predicate. Table 3 shows the details of such algorithms in which E_1 and E_2 denote two specific sets, $get(E_1)$ represents an element obtained from set E_1 if E_1 is a non-empty set (otherwise, it becomes undefined), and w is a natural number.

In this table, x_1 in Algorithms (1) and (2) denotes an *element* of a set (e.g., E_1), while in the rest of algorithms it denotes a set whose type is $set\ of\ T$ where T is the element type already defined. Algorithm (1) shows that a test case for x_1 can be generated by assigning any value from E_1 (as indicated by the assignment $x_1 := get(E_1)$). Algorithm (3) indicates that a test case for set x_1 to satisfy the condition $card(x_1) = w$ is to randomly select w elements from its element type T. Algorithm (4) states that a test case for set x_1 to satisfy the condition $union(x_1,\ E_1) = E_2$ (the union of x_1 and E_1 is equal to E_2) is to assign the difference set between E_2 and E_1 (i.e., $x_1 := E_2 \setminus E_1$). The other algorithms in the table can be similarly interpreted, we therefore do not elaborate on them for brevity.

Note that the algorithms in the table are suitable for dealing with expressions involving only one input variable x_1. However, if more than one input variable is involved, a similar measure to that of *Algorithm 1* can be taken.

Similarly, a group of algorithms are designed to deal with expressions involving operators on *sequences*. A sequence is a listing of ordered elements in which the duplication of elements is allowed. A sequence type contains a set of sequences; it can be declared in the format $seq\ of\ T$, where T, called element type, can be any type previously defined. These algorithms are summarized in *Algorithm 3* below.

Algorithm 3. *Let $q = 1$ and $Q(x_1, x_2, \ldots, x_q) \equiv E(x_1)$, where x_1 is a variable denoting a sequence of elements, $E(x_1)$ is an expression involving x_1 and an operator defined on sequence types, such as len (length of sequence), inds (indexes set of sequence), elems (element set of sequence), and conc (concatenation of sequences). Then, the algorithm for generating a test case from each kind of atomic predicate aims to find a sequence value that satisfies the predicate. Table 4 shows a set of test case generation algorithms for various atomic predicates in which E_1 denotes a set of natural numbers, E_2 a set value of appropriate type, E_3 and E_4 two specific sequences respectively, and w a natural number.*

In this table, algorithm (1) shows that a test case can be generated for sequence x_1 to satisfy the predicate $len(x_1) = w$ (the

TABLE 4
Algorithms for Test Case Generation From Sequence Type Expressions

No. of algorithms	Expression $E(x_1)$	Algorithms of test case generation for x_1	Algorithms for the other variables $(i = 2,...,n)$
(1)	$len(x_1) = w$	$x_1 := [a_1, a_2, ..., a_w]$ where x_1: *seq of* T $a_k \in T$, $k = 1, ..., w$	$x_i := any \in Type(x_i)$
(2)	$inds(x_1) = E_1$	$x_1 := [a_1, a_2, ..., a_w]$ where x_1: *seq of* T $a_k \in T$, $k = 1, ..., w$ $w = card(E_1)$	$x_i := any \in Type(x_i)$
(3)	$elems(x_1) = E_2$	$x_1 := [a_1, a_2, ..., a_w]$ where $a_k \in E_2$, $k = 1, ..., w$, and $\forall_{i,j \in [1..w]} \cdot i \neq j \Rightarrow a_i \neq a_j$	$x_i := any \in Type(x_i)$
(4)	$conc(x_1, E_3) = E_4$	$x_1 := [a_1, a_2, ..., a_k]$ where $E_3 = [a_{k+1}, ..., a_w]$, $1 \leq k < w$, and $E_4 = [a_1, a_2, ..., a_w]$	$x_i := any \in Type(x_i)$
(5)	$hd(x_1) = a_1$	$x_1 := [a_1, a_2, ..., a_w]$ where $a_k \in T$, $k = 2, ..., w$	$x_i := any \in Type(x_i)$
(6)	$tl(x_1) = E_4$	$x_1 := [a_1, a_2, ..., a_w]$ where $conc([a_1], E_4) = [a_1, a_2, ..., a_w]$	$x_i := any \in Type(x_i)$

TABLE 5
Algorithms for Test Case Generation From Composite Type Expressions

No. of algorithms	Expression $E(x_1)$	Algorithms of test case generation for x_1 whose field variables are a_1, a_2, ..., a_m declared with any types in SOFL	Algorithms of test case generation for the remaining variables $(i = 2,..., n)$
(1)	$x_1 = mk\_A(v_1, v_2, ..., v_m)$	$x_1.a_j := v_j, j = 1, ..., m$	$x_i := any \in Type(x_i)$
(2)	$x_1.a_j = v_j$	$x_1.a_j := v_j, j = 1, ..., m$	$x_i := any \in Type(x_i)$
(3)	$x_1 = modify(y,$ $a_1 \rightarrow v_1, a_2 \rightarrow v_2)$	$x_1 := y$; $x_1.a_1 := v_1$; $x_1.a_2 := v_2$;	$x_i := any \in Type(x_i)$
(3)	$x_1 = y$	$x_1 := y$	$x_i := any \in Type(x_i)$
(4)	$x_1 <> y$	$x_1 := y$; $x_1.a_1 := y.a_1 + \sigma$, $\sigma > 0$	$x_i := any \in Type(x_i)$

length of x_1 is w) by assigning x_1 a sequence of w elements taken randomly from the element type T. Algorithm (3) describes that a test case can be generated for sequence x_1 to meet the condition $elems(x_1) = E_2$ (the element set of x_1 is equal to set E_2) by assigning x_1 a sequence of elements covering all of the elements of E_2. Algorithm (5) states that a test case can be generated for sequence x_1 from the predicate $hd(x_1) = a_1$ by assigning x_1 a sequence of elements whose first element is a_1 and the rest are selected randomly from their type. Since the other algorithms in the table can be interpreted similarly, their elaboration is omitted.

Another important and frequently used data type is *composite type*. A composite type defines a set of composite objects, each being composed of several fields. Each field is represented by a variable that is declared with a certain type. The general format of a composite type declaration is illustrated by the type A declaration below:

$$\begin{aligned} A = \ & composed\ of \\ & a_1 : Type(a_1) \\ & a_2 : Type(a_2) \\ & \cdots \\ & a_m : Type(a_m) \\ & end. \end{aligned}$$

This specifies that each composite object of type A has m fields: $a_1,\ a_2, \ldots, a_m$. A group of algorithms for generating test cases from an expression involving operators on composite objects are summarized in *Algorithm 4*.

Algorithm 4. *Let $q = 1$ and $Q(x_1,\ x_2,\ \ldots,\ x_q) \equiv E(x_1)$, where x_1 denotes a composite object of type A, $E(x_1)$ is an expression involving x_1 and an operator defined on composite type, such as* make function *(e.g., $mk\_A(v_1, v_2, \ldots, v_m)$),* field select *(e.g., $x_1.a_1$), modify (e.g., $modify(mk\_A(v_1, v_2, \ldots, v_m)$, $a_1 - > newValue)$), equality $(=)$, and* inequality *$(< >)$. Then, a collection of algorithms for generating a test case from each kind of atomic predicate is shown in Table 5.*

Algorithm (1) in Table 5 suggests that a composite object x_1 of type A can be created to satisfy the equation $x_1 = mk\_A(v_1, v_2, \ldots, v_m)$ by assigning the designated values v_j $(j = 1, \ldots, m)$ to the corresponding field variables $x_1.a_j$. Algorithm (3) states that a composite object x_1 can be generated to meet the condition $x_1 = modify(y,\ a_1 - > v_1,\ a_2 - > v_2)$ by executing the three assignment statements sequentially: $x_1 := y$ (assigning composite object y to x_1), $x_1.a_1 := v_1$ (assigning value v_1 to the field variable a_1 of x_1), and $x_1.a_2 := v_2$. The other algorithms can be easily interpreted similarly, and their explanations are therefore omitted.

Since a test condition of a functional scenario can be a disjunction of several conjunctions of atomic predicates, the algorithms on atomic predicates discussed so far are not sufficient. We must discuss the issues in relation to test case generation from a conjunction and disjunction. Below in Section 3.2 we focus on conjunction and in Section 3.3 we discuss disjunction.

3.2 Generation From Conjunction

As indicated in *Criterion 2* of Section 2.2, the test condition $S_{pre} \wedge C_i$ of the scenario $S_{pre} \wedge C_i \wedge D_i$ $(i = 1, \ldots, n)$, and the scenario itself, can be transformed into an equivalent disjunctive normal form, say $P_1 \vee P_2 \vee \cdots \vee\ P_m$ $(m \geq 1)$, where each P_j $(1 \leq j \leq m)$ is a conjunction of atomic predicates, say $Q_j^1 \wedge Q_j^2 \wedge \cdots \wedge Q_j^w$ $(w \geq 1)$.

The essential idea of the algorithm for generating test cases from the conjunction is first to form an *ordered partition* of the atomic predicate set $\{Q_j^1,\ Q_j^2,\ \ldots,\ Q_j^w\}$ according to *predicate dependency*, and then properly apply the algorithms mentioned previously to generate a test case satisfying all of the atomic predicates in the conjunction if it is satisfiable. To comprehend the essential idea, we first need to clarify the concepts of predicate dependency and ordered partition.

Notation:

- $Var(E)$ denotes the set of free variables occurring in predicate E.
- $[1..n]$ denotes the set of integers $\{1, 2, \ldots, n\}$.
- $\{Q_j^1,\ Q_j^2,\ \ldots,\ Q_j^w\}$ denotes all of the atomic predicates in the conjunction $Q_j^1 \wedge Q_j^2 \wedge \cdots \wedge Q_j^w$.

Definition 13 (predicate dependency). *Let E_1 and E_2 be two predicate expressions. If $Var(E_1) \subset Var(E_2)$, E_2 is said to be dependent on E_1, which is represented as $E_1 \sqsubset E_2$.*

For example, predicate $x * y > 20$ is dependent on $x > 0$; that is, $x > 0 \sqsubset x * y > 20$.

Definition 14 (ordered partition). *Let $\{R_1,\ R_2,\ \ldots,\ R_u\}$ be a set of predicate sets. If it satisfies the following two conditions*

(1) $\forall_{i \in [1..u-1]} \forall_{E1 \in R_i} \exists_{E2 \in R_{i+1}} \cdot E1 \sqsubset E2$
(2) $\forall_{i \in [1..u]} \forall_{E1, E2 \in R_i} \cdot \neg(E1 \sqsubset E2)$,
we say $\{R_1,\ R_2,\ \ldots,\ R_u\}$ is an ordered partition on $\sqsubset$.

For instance, $\{\{x > 0,\ y > 1\}, \{x * y > 20\}, \{x + y * z > 1,\ x + y * z < 100\}\}$ is an ordered partition on $\sqsubset$, whereas

$\{\{x > 0,\ x + y > 1\}, \{x * y > 10\},\ \{x + y * z > 1,\ x + y * z < 100\}\}$ is not because there exists a predicate $x + y > 1$ in the first predicate set on which the predicate $x * y > 10$ in the second predicate set is not dependent (violating condition (1)). It is also because that the two predicates in the first predicate set satisfies the dependency relation, i.e., $x > 0 \sqsubset x + y > 1$, which violates condition (2) in the definition.

Definition 15 (predicate set satisfaction). *Let R be a predicate set and t be a test case. If t satisfies every predicate in R, we say t satisfies R.*

Suppose $R = \{x > 0,\ x * y < 10\}$ and a test case $t = \{(x, 2), (y, 3)\}$. Then, obviously t satisfies R by definition because t satisfies both $x > 0$ and $x * y < 10$.

The rational for forming the ordered partition based on the predicate dependency in the algorithm is that producing values for the variables in a predicate (e.g., $x + y * z > 1$) depends on the values already generated from the depended predicates (e.g., $x > 0$ and $y > 1$).

The essential idea of the algorithm to generate a test case satisfying the conjunction $Q_j^1 \wedge Q_j^2 \wedge \cdots \wedge Q_j^w$ is summarized as follows. First, construct $\{R_1,\ R_2,\ \ldots,\ R_u\}$ as an ordered partition on $\sqsubset$ for the set $\{Q_j^1,\ Q_j^2,\ \ldots,\ Q_j^w\}$ $(1 \leq u \leq w)$. Thus, $\{R_1,\ R_2,\ \ldots,\ R_u\}$ becomes an alternative expression of $\{Q_j^1,\ Q_j^2,\ \ldots,\ Q_j^w\}$ but more suitable for test case generation. Second, generate a test case that satisfies all the predicates in R_i $(1 \leq i \leq u)$ and then utilize the resultant test case to generate a more complete test case for R_{i+1} (i.e., produce values for more variables involved). Repeat this process until R_u is reached and a qualified test case is produced. However, if the generation fails for R_i, it will go one step back to retry generating a test case for R_{i-1} (provided that $i - 1 \geq 1$) and then repeat the same process. But if the number of failures to generate the qualified test case satisfying all $R_1,\ R_2,\ \ldots,\ R_u$ reaches a pre-defined one, a failure message will be issued as the result of the algorithm.

To generate a test case satisfying all the predicates in R_i, the algorithm generates a test case satisfying the first atomic predicate of R_i (assuming that the atomic predicates in R_i are arranged to appear from left to right) and then test whether it satisfies all of the other atomic predicates in R_i. If yes, a successful test case is generated; otherwise, repeat the same process for the other atomic predicates in R_i until all of the atomic predicates of R_i is exhausted. The algorithm to generate a test case satisfying the conjunction of $Q_j^1 \wedge Q_j^2 \wedge \cdots \wedge Q_j^w$ was reported in our previous conference paper [38] and the reader with further interest can refer to it for details.

One may argue that existing SAT or SMT solvers should be utilized to generate test data for conjunctions in our work. In fact, after a careful investigation, we realize that existing SAT solvers and SMT solvers aim to find a solution to tell whether a conjunction is satisfiable. Although they can be used for test case generation, requiring them to meet various test case generation criteria of our interest is difficult. Furthermore, SAT solvers, such as RISS [39], only deal with propositional logic, their capability is limited for our formal notation that adopts first-order predicate logic. Existing SMT solvers, such as Yices [40] and Z3 [41], may be a better possibility for the solution due to their capability of dealing with decidable fragments of predicate logic, but they are difficult to be adopted in our work because they do not cope with many compound types and the operators defined on those types provided in our specification language.

3.3 Generation From Disjunction

Compared to test case generation from a conjunction, algorithm for test case generation from the disjunction $P_1 \vee P_2 \vee \cdots \vee P_m$ $(m \geq 1)$ is rather simple. There are at least two kinds of solutions. One is to take a non-empty test set generated from any of the disjuncts P_j $(1 \leq j \leq m)$ as the resultant test set for the disjunction, but this is unlikely to exercise all of the disjuncts. A better choice is to take the union of all of the test sets generated from all of the disjuncts as the test set for the disjunction. Since the algorithm is trivial, we do not discuss its details here.

3.4 Motivation of V-Step

The *scenario-path coverage* described in *Criterion 1* indicates the adequacy for test set generation in our testing approach. It requires that not only all the functional scenarios in the specification be covered, but also all of the representative program paths of the program be traversed. Our discussions on algorithms for test case generation so far have focused on achieving a full coverage of functional scenarios in the specification, but those algorithms do not necessarily deal with the problem of how to gain a full coverage of the representative program paths in the corresponding program. As is well-known in testing, bugs on a path is unlikely to be revealed if the path is never executed.

To address this concern, more test cases need to be produced if the already generated test set does not ensure an expected coverage of the paths. The challenge is what and how test cases should be generated so that representative paths of the program can all be traversed. It is worth understanding that this challenge is not limited to our testing approach but a common problem to all of existing specification-based testing approaches.

In this section, we describe a *"Vibration"* step, called *V-Step*, as one-step forward to this problem. The V-Step is an integral part of our V-Method, and presents a technique for utilizing the test case generation algorithms introduced previously to produce adequate test cases meeting *Criterion 1*. Future development of the technique is expected to lead to more effective and efficient solutions. Our V-Step focuses on test case generation from an atomic predicate, which is a fundamental problem because test case generation from conjunctions and disjunctions depends on it.

The V-step is basically a heuristic method of local search for new test data, depending on a notion of "distance". A simple example helps to illustrate the essential idea of the technique. Let $E_1 > E_2$ be a relation between two expressions E_1 and E_2. Taking the V-Step we will first produce values for all the variables involved in both E_1 and E_2 such that $E_1 > E_2$ holds and the distance between E_1 and E_2, i.e., $|E_1 - E_2|$ (absolute value), is treated as the base distance, and we will then create a new test case satisfying the relation when the distance is changed deliberately greater or smaller (see a detailed discussion of distance in Section 3.6) according to a certain strategy. Repeating this process by increasing and decreasing the distance between E_1 and E_2, respectively, until a decision for

TABLE 6
Summary of the Example for the V-Step

Test data	Distance	Distance Up	Distance Down	Decision Evaluation
$x = 100$, $y = -100$	300			$x < y + 100$ (F) $x >= y + 100$ && $x < y + 300$ (T)
$x = 50$, $y = 0$	450	150		$x < y + 100$ (T)
$x = 160$, $y = -150$	190		260	$x < y + 100$ (F) $x >= y + 100$ && $x < y + 300$ (F) $x >= y + 300$ && $x < y + 500$ (T)

terminating the test case generation is made. It is worth mentioning that the notion of distance between two numeric test cases was used in the work of Chen *et al.* on adaptive random testing (ART) [42] and extended to objects in the work of Ciupa *et al.* on adaptive random testing for object-oriented software (ARTOO) [43], [44], but in our work, the distance is defined between two formal expressions in a relation from which test cases are generated.

The rational for the V-Step comes from the theory that a functional scenario is usually refined into a program fragment that may implement the defining condition of the scenario conditionally. This point can be understood easily with an example. Assume $x < y + 500 \wedge z = x + y$ is a single functional scenario in which $x < y + 500$ is the test condition involving only input variables x and y, and $z = x + y$ the defining condition involving output variable z. This scenario can, for example, be refined into the following conditional statement in the program:

```
1   if (x < y + 100) {
2    i = i + 1;
3    z = x + y;
4   } else if (x >= y + 100 && x < y + 300) {
5      i = i + 2;
6      z = x + y;
7      } else if (x >= y + 300 && x < y + 500){
8        i = i + 3;
9        z = x + y;
10     }.
```

In any branch of the nested conditional statement, the defining condition is correctly implemented, but meanwhile for some reason the program also needs to update another variable i , which can be either a local or "global" variable, differently in different branches.

To make each decision true and false at least once, respectively, different test cases changing within the range $x < y + 500$ need to be used. For example, let $x = 100$ and $y = -100$ as the first test data. The distance between the two expressions x and $y + 500$ is 300. This test data makes the first decision $x < y + 100$ false and the second decision $x >= y + 100$ && $x < y + 300$ true, thus covering the path [1,4,5,6]. Next, we enlarge the distance between x and $y + 500$ to 450, and generate the new test data $x = 50$ and $y = 0$. It makes the first decision $x < y + 100$ true, thus covering the path [1,2,3]. Another change is to decrease the distance 450 by 260 to 190, and generate the test data $x = 160$ and $y = -150$. This makes the first decision $x < y + 100$ false, the second decision $x >= y + 100$ && $x < y + 300$ false, and the third decision $x >= y + 300$ && $x < y + 500$ true, thus covering the path [1,4,7,8,9,10]. By the above test, all of the paths in the program fragment implementing the functional scenario are traversed. Table 6 shows a summary of the most relevant information of the test for readability.

3.5 Principle of V-Step

Let $E_1(x_1, x_2, \ldots, x_n)\ R\ E_2(x_1, x_2, \ldots, x_n)$ denote that expressions E_1 and E_2 have relation R (e.g., $<$, $>$). Applying the V-Step to this relation, we first assign some values to $x_1, x_2, \ldots, x_n$ such that the relation $E_1(x_1, x_2, \ldots, x_n)\ R\ E_2(x_1, x_2, \ldots, x_n)$ holds with an initial distance between E_1 and E_2, and then repeatedly create further values for the variables such that the relation still holds but the distance between E_1 and E_2 "vibrates" (changes repeatedly according to the distance) between the initial distance and the "maximum" or "minimum" distance, until a decision for terminating the testing is made. The decision is made based on one of the two conditions. One is that all the related program paths have been traversed, and the other is that a required number (usually pre-defined) of test cases are generated. The former indicates that the test cases have exercised all the related paths, while the latter implies that covering all the related paths is difficult (possibly due to the existence of bugs or paths not executable).

An algorithm of the V-Step can be realized by an operation (e.g., a method in a Java class) to which three formal parameters $noOftestcase$, $distanceUp$ and $distanceDown$ are needed. $noOftestcase$ is the maximum number of test cases to be generated before a newly traversed path is found; it is used to terminate the algorithm, indicating the impossibility of finding any new path even if more test cases are generated. $distanceUp$ is used to increase the distance between the two expressions and $distanceDown$ is used to decrease the distance. The update of the distance is done by a $Distance$ function in the algorithm. Both $distanceUp$ and $distanceDown$ are positive integers and their actual values are determined by the calling operations (e.g., methods in Java), according to the desired degree of changing the distance. The desired degree can be decided by the tester, but it can also be automatically set.

When all the variables involved in the relation are of numeric types, such as integers or real numbers, a general algorithm for automating this process can be simple, but challenge will arise when the types are more complex compound types, such as set, sequence, or composite types adopted in most of the model-based formal notations. The core of the challenge lies in the difficulty in defining the concept of distance between the two expressions E_1 and E_2 if the relational operator is not for comparison between numeric values but something else, such as $inset$ (membership) or $subset$. In the next section, we will discuss the issue of how the distance can be achieved by the $Distance$ function mentioned above.

3.6 Distance Definitions on Types

In the discussion of this section we use E_1 and E_2 to represent two expressions of generic types, but when the $Distance$ function is defined on a specific type (e.g., int), the types of E_1 and E_2 will be automatically specialized to that type, unless we clearly mention it. We will define the

TABLE 7
Definition of the Distance Function on Numeric Types

$Distance$ **function definition (for** $nat0$**,** nat**,** int**,** $real$ **types)**
$Distance(E_1, E_2, "R") \equiv abs(E_1 - E_2)$ where $R \in \{=, <, <=, >, >=, <>\}$

function on numeric types, including $nat0$ (natural numbers including zero), nat (natural numbers without zero), int, and $real$, $char$ (characters), enumeration types, and set types, respectively. Basically, a distance between two expressions E_1 and E_2 is the "difference" between the two values resulting from the evaluation of the two expressions, indicating how "far" it is between the two values. The core of defining the concept is how to interpret the "difference" . As discussed next, the "difference" between two values is defined differently on different types, but all of them reflect our perception of difference in some view.

The definition of the $Distance$ function on numeric types is given in Table 7. The definitions of the $Distance$ function on the character type and enumeration types are given in Table 8. In all these definitions, when relational operator is "$=$", the distance between the two expressions is defined as 0; otherwise in the case of $char$, it is defined as 1 (indicating the least "difference"), and in the case of enumeration types, the distance is defined as the absolute value of the difference between the indexes of the two operands. Note that when the distance is 0, the *V-Step* still generates test data that satisfy the related equation. For example, from the equation $E_1 = E_2$ in Table 8, we can generate $E_1 =$ 'a' and $E_2 =$ 'a', $E_1 =$ 'q' and $E_2 =$ 'q', respectively.

The definition of the $Distance$ function on set types are given in Table 9. In this definition, we treat a set as an indexed set in the sense that each element of the set is indexed by a natural number. Let $index(a, P)$ be an index function that yields the index of element a in set P. In the case of set $P = \{2, 5, 9, 10\}$, for example, we have $index(2, P) = 1$, $index(5, P) = 2$, $index(9, P) = 3$, and $index(10, P) = 4$. This treatment of sets allows us to easily define the distance between two elements in a given set. Moreover, the distance between two sets E_1 and E_2 with respect to the relation $subset(E_1, E_2)$ (i.e., subset) is defined as the difference between their cardinalities (denoted by $card(E_1)$ and $card(E_2)$); and likewise for the distance between two sets with respect to the relation $psubset(E_1, E_2)$ (i.e., proper subset). Note that $Distance(E_1,\ E_2,\ "="\ ) \equiv 0$ only means that the distance between E_1 and E_2 is zero, but it does not mean that the set variables involved in expressions E_1 and E_2 can not take different values in the next test case generation. For example,

TABLE 8
Definition of the Distance Function on the Other Basic Types

Types	$Distance$ **function definition**
$char$	$Distance(E_1, E_2, "=") \equiv 0$ $Distance(E_1, E_2, "<>") \equiv 1$
$enumeration$	$Distance(E_1, E_2, "=") \equiv 0$ $Distance(E_1, E_2, "<>") \equiv$ $abs(index(E_1) - index(E_2))$ where $index(E_1)$ denotes the index of the value E_1 in the $enumeration$ type; and likewise for $index(E_2)$.

TABLE 9
Definition of the Distance Function on Set Types

$Distance$ **function definition for set types**
$Distance(E_1, E_2, "subset") \equiv card(E_2) - card(E_1)$
$Distance(E_1, E_2, "psubset") \equiv card(E_2) - card(E_1)$
$Distance(E_1, E_2, "inset") \equiv card(E_2) - index(E_1, E_2)$
$Distance(E_1, E_2, "notin") \equiv card(E_2)$
$Distance(E_1, E_2, "=") \equiv 0$
$Distance(E_1, E_2, "<>") \equiv abs(card(E_1) - card(E_2))$

suppose $x = y$ is a relation describing that integer sets x and y are the same. When generating test cases, we first generate $x = \{-1, 0, 1\}$ and $y = \{-1, 0, 1\}$, and then generate $x = \{-10, 0, 10\}$ and $y = \{-10, 0, 10\}$, and the third would be $x = \{-100, 0, 100\}$ and $y = \{-100, 0, 100\}$. Similarly, the $Distance$ function can also be defined on sequence types, map types, and composite types, which are adopted in SOFL and many other model-based formal notations. Since the definition shares the same principle for defining the function on set types, we omit the detailed discussion for brevity.

Note that the above definitions of the function $Distance$ on various types present only one way to compute the distance between the two sides of a relation. This does not imply that other ways of defining the distance function are impossible. Each way of defining the $Distance$ function determines the style of a specific "vibration" in selecting test cases. We have realized through our experience that it would be difficult to define the $Distance$ function for obtaining an efficient path coverage without taking advantage of the program structure, and plan to investigate more complex grey-box approaches in our future work.

3.7 Prototype Tool

We have developed a prototype tool that implements the above algorithms for test case generation based on SOFL process specifications. Despite the fact that the latest version of the tool is limited in dealing with deeply nested data structures (e.g., set of sequence of composite objects) and handling large scale applications, it has allowed us to demonstrate the feasibility of implementing the algorithms. Basically, the tool supports automatic test case generation from a single functional scenario of a process specification and the evaluation of the defining condition provided that the final value of the corresponding output variable is supplied. Fig. 3 shows a snapshot of the tool generating a test case from a process specification. Since the work on the supporting tool is beyond the scope of this paper, we omit the detailed discussion concerning it here. As pointed out in Section 7, we will continue to work on the improvement of this tool and will integrate it with other related prototype tools in our future work.

4 Controlled Experiment

As is well-known, experiments in software testing are usually costly and difficult to carry out due to many constraints on resources. As a research lab in the university environment, we could take advantage of our graduate students to help conduct experiments, as many other research groups do in academia [45]. The students have experienced our testing method and the related testing approaches; they are assigned to prepare the testing target programs, generate test cases, and perform the actual testing of the programs in

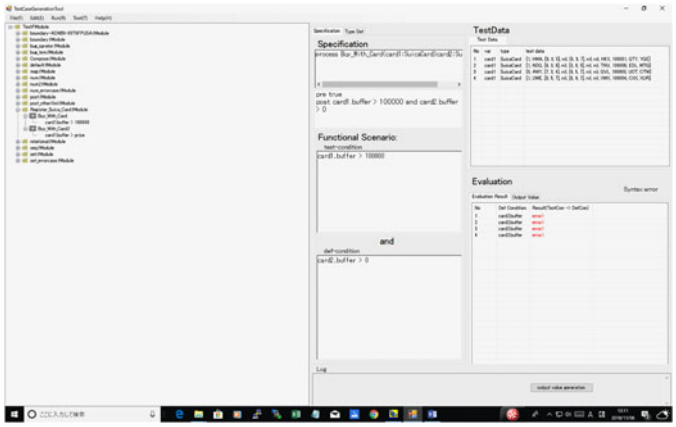


Fig. 3. Snapshot of the SOFL prototype test tool.

our experiment (see details in Section 4.2). The experiment compares our V-Method with the compatible *Functionality-Based Input Domain Partition Method* (FBIDPM) [31], [46] using mutation testing in which the mutants are produced from the original code of a Universal Card System (UCS). The idea of the FBIDPM is to identify characteristics that correspond to the intended functionality of the program under test and to generate test cases based on the functionality-based characteristics. In our experiment, we use the pre- and post-conditions in the specification of an operation as the sources for the characteristics, but focus on the conditions that help partition the input domain in test case generation. The comparison of the two methods is mainly measured in terms of mutation score since it is a commonly used criterion [47].

We describe the experiment below by elaborating on the system background, experiment setup, experiment result, threats to the validity, and discussions on potential challenges of our V-Method.

4.1 System Background

The system used for this experiment is called *Universal Card System* (UCS), which is developed to support the future use of the *universal card* (UC) in Japan. The universal card is intended to be a unified card that can be used to enjoy the railway services, automated teller machine (ATM) services of banks, shopping at supermarkets, and specified library services. SOFL was used to write the formal specification of UCS that contains 1,141 lines of formal expressions. C# was used for implementation of the software that contains 7,422 lines of code.

4.2 Experiment Setup

In order to reduce the impact of the human factor on the result of the experiment, using multiple tests conducted by different groups is desirable. We generate two test suites manually by different groups using our V-Method and the FBIDPM, respectively. We use TEST1 and TEST2 to represent the two tests using the two test suites, respectively. Further, in TEST1, we use TEST1-V to name the test carried out with the test cases generated using the V-Method and TEST1-DP (domain partition) the test carried out with the test cases generated using the FBIDPM. Similarly, in TEST2, we use TEST2-V to represent the test with the test cases generated using the V-Method and TEST2-DP the test with the test cases generated using the FBIDPM. To avoid unnecessary mistakes in test result evaluations, we ensure that the result of a test is always judged under the monitoring of all the relevant persons, including the test case generator, the test performer, and the programmer of the UCS.

4.2.1 TEST1

The goal of TEST1 is to compare the two testing methods by following their principle for generating test cases. Both the V-Method and the FBIDPM generate test cases based on the domain partition defined by the pre- and post-conditions of an operation, but they may generate different number of test cases. In general, since the "vibration" algorithm in the V-Method is used if the current test case explores a new program path, the V-Method tends to produce more test cases than the FBIDPM that requires only selecting a representative test case for each subset in the domain partition. In TEST1, we intend to observe the difference between the two methods in such a natural manner.

The test suite for TEST1 is generated manually by the first author of this paper using the two different methods in the manner mentioned above, respectively. This test set is applied to test the mutants of the original code of UCS. An experienced researcher in our research group is employed to carry out the injection of bugs to the original programs of UCS to create the mutants manually. Each mutant is a faulty program, which is produced by inserting one bug into the code of a method (i.e., a subroutine) in a C# class. A number of different mutants are usually created from the same original method by means of a systematic bug injection that ensures an even distribution of different kinds of bugs over the program paths. We used 13 kinds of mutant operators for TEST1, including *arithmetic operator replacement*, *variable replacement*, *logical operator replacement*, *exception handling change*, and *statement position change*.

The actual testing (i.e., using the generated test cases to run the mutants using the Microsoft Visual Studio) is carried out by a graduate student who has around five years experience in SOFL formal specification techniques and about three years experience in using the testing methods. For each mutant, the test cases generated with our method and the test cases generated with the FBIDPM are used separately to test the mutant. Each test result is analyzed by the student together with the programmer of UCS based on the test oracle derived from the related specifications. Specifically, the student tester compares the execution result (output values) with the test oracle if it is given as the expected output values. If both values are not consistent, the tester will tell that the mutant is killed and this judgement will be confirmed by the programmer. But if the test oracle is given as a logical expression, as we mentioned in Section 2.3 , then the tester will have to substitute the output values of the execution for the corresponding output variables in the logical expression and evaluate it. According to this result, the tester will tell whether the mutant is killed or not. Again, in this case the programmer plays the role of confirming the conclusion made by the tester.

4.2.2 TEST2

The goal of TEST2 is to compare the two testing methods by using the same number of test cases generated using each of

TABLE 10
The Result of the Test TEST1-V

Processes	No. of test cases	No. of mutants created	No. of mutants killed	MS (%)
MUC_Card	6(2)	10	10	100
UTF_Table	7(2)	6	6	100
UUC_Type	8(8)	5	4	80
UC_Amounts	8(4)	7	7	100
CUC_Card	6(6)	5	4	80
SA_Balance	8(3)	10	9	90
Withdraw	9(6)	15	15	100
Deposit	10(10)	10	8	80
Transfer	10(10)	13	11	85
B_Dollars	7(7)	10	9	90
B_JPY	7(6)	10	10	100
B_Book	6(1)	8	8	100
RT_Book	6(4)	10	10	100
S_Book	9(3)	6	6	100
RN_Book	9(9)	10	7	70
CF_Overdue	6(6)	7	6	86
CC_WC	8(8)	10	7	70
CC_FBA	8(8)	18	14	78
R_Money	7(7)	10	9	90
BR_Ticket	8(5)	12	12	100
REN_Control	10(5)	10	9	90
REX_Control	10(7)	10	10	100
CUC_Card	10(4)	10	10	100
M_Payment	4(2)	9	9	100
C_Payment	8(5)	7	7	100
Total	**195(138)**	**238**	**217**	**91**

the two methods, under the condition that the principles of both methods are properly applied in the test case generation. This allows us to observe the effect of the V-Method without any bias potentially to be caused by using different number of test cases generated by different testing methods. However, the difficulty is that the number of test cases generated by the V-Method is usually greater than that produced by the FBIDPM because the V-Method requires to use the "vibration" algorithm to make more test cases for the exploration of more program paths whereas the FBIDPM only requires the selection of a representative test case from each partitioned sub-domain. In TEST2, we adopt the following approach to deal with this problem. To test each mutant, we strictly apply the principle of the V-Method and the FBIDPM to produce two test sets, respectively. If any test set, say T_1, has less number of test cases than the other, say T_2, we then randomly generate more test cases for test set T_1 to ensure that it has the same number of test cases as T_2. Our experience in the experiment shows that in all circumstances, the number of test cases generated by the FBIDPM is smaller than that of the test cases generated by the V-Method. To make the numbers equal, we produce additional test cases randomly for the test set produced by the FBIDPM. Since using more test cases in this context will not damage the effect of the FBIDPM in terms of killing mutants (instead, its effect might be strengthened because additional test cases may help kill more mutants for the FBIDPM), the result of the comparison between the two methods cannot be in favor of our V-Method.

TEST2 is carried out by a different group of graduate students from those conducting TEST1. The test suite of TEST2 is produced manually by a well-experienced graduate student in our lab. This test suite is applied to another set of mutants of the original code of UCS. A different student, who was working on mutation testing, carried out the injection of bugs to the original programs of UCS to create the mutants in the same way as for TEST1 but with a slightly different set of mutant operators. The operators include *arithmetic operator replacement*, *relational operator replacement*, *logical operator replacement*, *constant replacement*, *assignment statement change*, *assignment statement deletion*, *condition negation*, *unary operator change*, *iteration statement change*, *statement block deletion*, and *method invocation change* . It is worth mentioning that we could use the same set of mutants created for TEST1 for the experiment in TEST2, but since that will pose a threat to the validity of the experiment, we did not take that approach. The threat is that the same set of mutants may be sensitive only to either the V-Method or the FBIDPM. Using two different sets of mutants as we have adopted in our experiment can avoid this threat.

The actual testing for TEST2 is done using the Microsoft Visual Studio by another student who has been working on software testing based on SOFL formal specifications for around three years. For each mutant, the test cases generated using our method and the test cases generated using the FBIDPM are used separately to test the mutant. Each test result is analyzed by checking whether the result satisfies the test oracle derived from the related specification and comparing the result of running the mutant with that of executing the corresponding original program with the same test case.

4.3 Experiment Results

As indicated in Section 2.1, in a SOFL formal specification, a module may contain several processes and each process may contain several functional scenarios. Since process is an independent basic functional unit, we present the experiment results at the process level. This also allows us to present the experiment result comprehensibly because it avoids uninteresting details that may affect the readability.

Table 10 shows the result of TEST1-V and Table 11 gives the result of TEST1-DP. In the tables, the original process names used in the formal specification are properly simplified for the sake of space. For each process, we present the number of test cases generated for running the mutants, together with the maximum number of the test cases that are actually needed to kill the mutants (in the parenthesis), the number of the mutants created and the number of the mutants killed. We also show the *mutation score* (MS) that is the ratio between the number of the mutants killed to that of the mutants created. For instance, in testing the mutants of the process MUC_Card using the V-Method, we used 6 test cases in which 2 test cases were used to kill the mutants, created 10 mutants, killed 10 mutants, and produced the 100 percent MS.

The result of TEST1 shows that the test cases generated using the V-Method kill more mutants than those produced using the FBIDPM for some processes but kill the same number for some other processes. The average mutation score for the V-Method is approximately 91 percent, higher than 76 percent for the FBIDPM.

Tables 12 and 13 show the result of TEST2-V and TEST2-DP, respectively. The format of these two tables is the same as that of Tables 10 and 11 except that we also present the path coverage information collected manually by the tester

TABLE 11
The Result of the Test TEST1-DP

Processes	No. of test cases	No. of created mutants	No. of killed mutants	MS (%)
MUC_Card	2(2)	10	9	90
UTF_Table	2(2)	6	4	67
UUC_Type	3(3)	5	4	80
UC_Amounts	3(1)	7	7	100
CUC_Card	3(3)	5	3	60
SA_Balance	2(2)	10	8	80
Withdraw	4(4)	15	12	80
Deposit	3(3)	10	7	70
Transfer	4(4)	13	10	77
B_Dollars	3(3)	10	6	60
B_JPY	3(3)	10	9	90
B_book	3(3)	8	7	88
RT_Book	2(2)	10	6	60
S_Book	3(3)	6	5	83
RN_Book	3(3)	10	7	70
CF_Overdue	3(2)	7	6	86
CC_WC	3(3)	10	7	70
CC_FBA	4(4)	18	13	72
R_Money	2(2)	10	8	80
BR_Ticket	3(3)	12	10	83
REN_Control	3(3)	10	5	50
REX_Control	3(3)	10	6	60
CUC_Card	5(3)	10	10	100
M_Payment	2(2)	9	7	78
C_Payment	1(1)	7	5	71
Total	**72(67)**	**238**	**181**	**76**

in TEST2. For instance, in testing the mutants of the process MUC_Card using the V-Method, we cover all the 6 paths out of the total 6 paths.

TEST2 exhibits a consistent result with that of TEST1, showing the superiority of the V-Method over the FBIDPM. TEST2-V achieves the 95 percent average mutation score and the 60 percent average path coverage rate with 1,800 test cases, whereas TEST2 obtains the 69 percent average mutation score and the 35 percent average path coverage rate with the same number of test cases.

We have also collected the data of various respects for comparison based on both the results of the two tests. Fig. 4 shows the box plots comparing the MS performance of the two methods in TEST1, TEST2, and both for all of the processes involved. In the case of TEST2, we have collected the path coverage information for each process. Fig. 5 shows the box plots comparing the path coverage rate obtained by the two methods in TEST2 for all of the processes, where the path coverage rate is defined as follows:

$$path\ coverage\ rate = the\ number\ of\ paths\ traversed\ /\ the\ total\ number\ of\ paths.$$

4.4 Threats to Validity

As is well-known, any experiment in software engineering faces threats to its validity and our experiment is not an exception. In this section, we discuss how we made efforts during the process of setting up and carrying out our experiment to mitigate the major threats to its validity.

4.4.1 Non-Determinism of Algorithms

The two testing methods used in our experiment are all non-deterministic in producing test cases, and therefore the test cases produced by any of them may vary randomly within a scope. To evaluate the performance of these algorithms, it is necessary to repeatedly use each algorithm in a relatively great number of times. In TEST1, the V-Method is applied to

TABLE 12
The Result of the Test TEST2-V

Processes	No. of test cases	No. of created mutants	No. of killed mutants	MS (%)	No. of covered paths (total), percentage
MUC_Card	44(12)	11	11	100	6(6), 100%
UTF_Table	28(5)	5	5	100	4(4), 100%
UUC_Type	29(6)	6	6	100	6(6), 100%
UC_Amounts	32(5)	6	5	83	4(5), 80%
CUC_Card	8(2)	2	2	100	2(2), 100%
SA_Balance	59(15)	9	9	100	5(6), 83%
Withdraw	103(31)	21	21	100	42(68), 62%
Deposit	83(24)	19	17	89	32(44), 73%
Transfer	305(93)	40	38	95	87(276), 32%
B_Dollars	164(51)	24	23	96	36(44), 82%
B_JPY	193(66)	28	27	96	41(44), 93%
B_Book	52(18)	9	8	89	5(6), 83%
RT_Book	42(9)	7	7	100	7(7), 100%
S_Book	10(4)	2	2	100	3(3), 100%
RN_Book	64(12)	8	7	88	6(7), 86%
CF_Overdue	21(6)	5	5	100	3(3), 100%
CC_WC	38(12)	9	8	89	7(8), 88%
CC_FBA	145(47)	22	21	95	19(20), 95%
R_Money	56(15)	14	12	86	12(14), 86%
BR_Ticket	75(37)	10	10	100	7(7), 100%
REN_Control	18(4)	4	4	100	4(4), 100%
REX_Control	25(7)	6	6	100	4(5), 80%
CUC_Card	126(36)	17	15	88	10(11), 91%
M_Payment	48(16)	10	10	100	9(9), 100%
C_Payment	32(8)	8	7	88	7(8), 88%
Total	**1800(541)**	**302**	**286**	**95**	**368(617), 60%**

TABLE 13
The Result of the Test TEST2-DP

Processes	No. of test cases	No. of created mutants	No. of killed mutants	MS (%)	No. of covered paths (total), percentage
MUC_Card	44(8)	11	8	73	4(6), 67%
UTF_Table	28(7)	5	5	100	4(4), 100%
UUC_Type	29(3)	6	3	50	3(6), 50%
UC_Amounts	32(4)	6	4	67	3(5), 60%
CUC_Card	8(2)	2	2	100	2(2), 100%
SA_Balance	59(13)	9	9	100	5(6), 83%
Withdraw	103(53)	21	16	76	22(68), 32%
Deposit	83(22)	19	13	68	15(44), 34%
Transfer	305(74)	40	28	70	36(276), 13%
B_Dollars	164(43)	24	18	75	22(44), 50%
B_JPY	193(45)	28	21	75	31(44), 70%
B_book	52(5)	9	5	56	4(6), 67%
RT_Book	42(4)	7	4	57	4(7), 57%
S_Book	10(4)	2	2	100	3(3), 100%
RN_Book	64(7)	8	6	75	4(7), 57%
CF_Overdue	21(4)	5	4	80	2(3), 67%
CC_WC	38(6)	9	5	56	4(8), 50%
CC_FBA	145(25)	22	15	68	14(20), 70%
R_Money	56(7)	14	7	50	7(14), 50%
BR_Ticket	75(27)	10	6	60	4(7), 57%
REN_Control	18(6)	4	2	50	2(4), 50%
REX_Control	25(2)	6	2	33	2(5), 40%
CUC_Card	126(33)	17	13	76	8(11), 73%
M_Payment	48(5)	10	5	50	5(9), 56%
C_Payment	32(4)	8	4	50	4(8), 50%
Total	**1800(413)**	**302**	**207**	**69**	**214(617), 35%**

produce 195 test cases that lead to approximately 514 runs of the mutants. These test cases are produced by following the principle of the V-Method. The result of the test shows that only 138 out of the 195 test cases are actually needed to kill the 217 out of the total 238 mutants. In the meanwhile, the FBIDPM is applied to produce 72 test cases that are used to run the mutants approximately 392 times. These test cases are produced by following the principle of the FBIDPM without any other constraints. Only 67 out of the 72 test cases are actually needed to kill 181 out of 238 mutants.

In TEST2, the V-Method is applied to generate 1,800 test cases that lead to approximately 3,547 runs of the mutants. As mentioned in Section 4.2, these test cases are produced in an interactive manner. Specifically, to test each mutant, we first use the V-Method to produce a test case from each functional scenario of the operation specification to run the mutant, and then apply the "vibration" algorithm to generate more test cases from each functional scenario to test the mutant until no more new program path is found by the recent three test cases. The test result shows that only 541 test cases are needed to kill 286 out of 302 mutants. After finishing the test using the V-Method, we carry out the test using the FBIDPM in TEST2. 1,800 test cases are produced to carry out approximately 3,547 runs of the mutants in which 413 test cases are needed to kill 207 out of 302 mutants.

Since the V-Method includes many small algorithms dealing with the generation of test cases from different kinds of predicate expressions, it is extremely difficult for

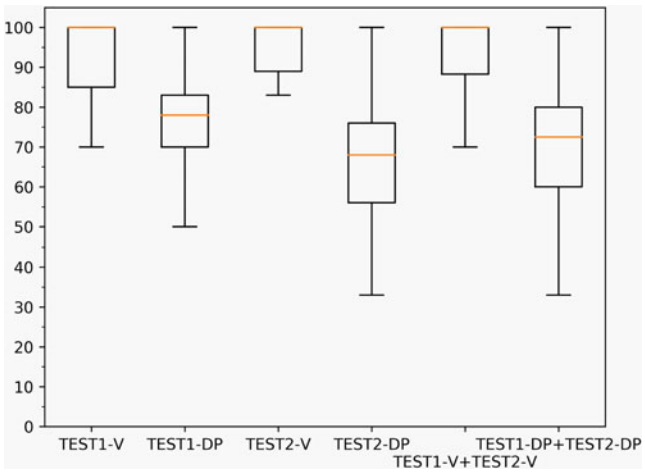


Fig. 4. Comparison of the MS performance of the two methods in TEST1, TEST2, and both.

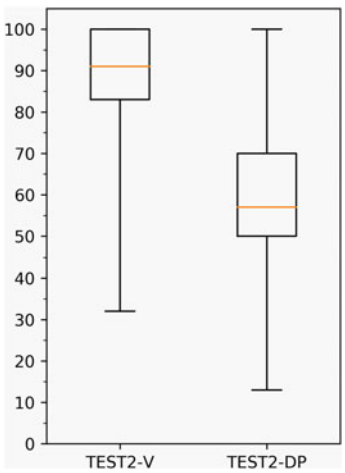


Fig. 5. Comparison of path coverage rate of the two methods in TEST2.

us to ensure that each of them is applied in a rather great number of times in the experiment. We believe that carrying out a much larger scale of experiment will be extremely useful in accurately evaluating the algorithms, and we plan to do so in our future work when the conditions (e.g., availability of a large program code and its formal specification, sufficient number of well trained subjects, sufficient budget) for such an experiment are made ready.

4.4.2 Number and Complexity of Used Target Programs

Another threat to the validity of our experiments is concerned with the number and complexity of the target systems used for testing. In general, the more target systems are used, the more reliable the experiment results are. To mitigate this threat, within our capability of affordable time, resources, and budget, we have chosen 25 processes and their programs of 5 application systems covering 5 different domains in the UCS as the target systems for testing in our experiment. The time complexity of most programs used is $o(n^2)$ (quadratic time complexity). These numbers may not be big enough to ensure a profound evaluation of our method, but it does allow us to observe and analyze its performance in a relatively small scale experiment for scientific study. Larger scale empirical studies are required for more comprehensive evaluations of our method in the future.

4.4.3 Human Factors

As is well-known, human factors are the major threat to the validity of experiments generally in software engineering [48]. As far as our experiments are concerned, there are two specific aspects that might be influenced by human factors. One is concerned with test case generation and the other is concerned with the judgement of whether a bug is found (or a mutant is killed) by a test.

To reduce the human influence in test case generation, the test suite for TEST1 is generated by the first author of this paper and the test suite for TEST2 is produced by a graduate student in our lab. Both are well experienced in using the two testing methods employed in our experiment and can ensure that each testing method is applied to generate test cases in accordance with its principle and rules, though the process can hardly be guaranteed to be perfect.

To reduce the human influence in the bug discovery judgement, the testers and the programmer work together to make judgements on bug discovery based on the derived test oracles. The judgements are finally confirmed and corrected (if applicable) through a session of face to face analysis and discussions by the test case generators, the persons who inserted bugs, the testers, and the programmer of UCS. Each person looks at the result from a different view and therefore all of the different views help the tester make the final judgement as accurately as possible.

4.4.4 Weakness of Mutation Testing

Our method is evaluated using mutation testing [49], [50], which has become a standard approach to evaluating the quality of test data. However, as pointed out in the literature [51], [52], [53], using equivalent mutants and/or subsumed mutants (or redundant mutants) is likely to distort the effect of the mutation testing. As pointed out in [53], typically equivalent mutants and subsumed mutants are machine-generated by a mutation testing tool. To mitigate this kind of threat, in our experiment, we manually create the mutants as mentioned in Tables 10, 11, 12, and 13 during which we have tried to avoid equivalent mutants and subsumed mutants. To avoid equivalent mutants, the mutant creators first apply a selected mutant operator to statements and/or conditions in the program and then carry out a dependency analysis in the related program segment, as suggested in [54], to confirm that the created mutant will result in a behavior different from the original program. To avoid subsumed mutants, the mutant creator always tries to apply different kind of mutant operator to different part of the statements or conditions on program paths. The resultant mutants are reviewed by the first author of this paper to ensure the quality of the mutants, in particular the avoidance of equivalent mutants and subsumed mutants. The results of the two tests TEST1 and TEST2 have also been analyzed by the mutant creators and the first author together to confirm that all of the killed mutants are not equivalent or redundant.

4.5 Discussion

Despite the effectiveness of our V-Method indicated in our experiment above, we have also learned several challenges to the V-Method for automatic program testing through our research. First, it seems extremely difficult, if not impossible, to establish a specification-based test case generation method that could guarantee a full path coverage of any given program. The challenge lies in the fact that a formal specification might be abstract, sometimes even incomplete, and generally has infinite ways of implementation. Our V-Method provides a "directional" solution to this problem but still lacks a solid theory for the guarantee. Considerable efforts are underway in our research lab for further investigation in this area.

Second, if an atomic predicate is not a linear expression, such as $x^5 + y^{10} \times z^2 > w^2 + z^3$, automatic generation of a test case from it would be extremely difficult. The hardest thing is the formation of a general algorithm that can deal with all kinds of such expressions with a practically acceptable efficiency. A similar difficulty also exists in dealing with some set expressions, such as $x \notin E$ where E is a very large or infinite set. Fortunately, our experience with many practical software development projects suggest that the complicated situations like the above hardly appear in practice. Therefore, this challenge may mainly remain in theory.

Third, to make the automatic testing technique effective, the implemented program must preserve the signature of its specification (i.e., all input, output, external variables of both the specification and the program must be the same, although their types can be in different format). However, this seems difficult to guarantee in practice, although theoretically the consistency can be achieved through configuration management. One way to ensure this $1:1$ relation is through *automatic signature preserving transformation* from specifications to programs. Another possibility is to strongly enforce the discipline that the program is implemented with

the assurance of keeping its module interface consistent with the signature of the specification.

Finally, when using the test cases generated based on the specification syntax to test the programs, the test cases must be translated into the program language suitable for executing the program. While such a translation can be rather easily done by humans who are familiar with both the specification notation and the program language, it may require a considerable support in a software tool. The reason is that an abstract type of an input variable in the specification (e.g., set type) can be implemented by one of the several different concrete types (e.g., array, vector, list) in the program and such an implementation is unknown in advance. If the test cases need to be automatically translated into the program syntax suitable for running the program, the tool must prepare all of the possible solutions to deal with every possible choice of the concrete data structure.

5 Related Work

Since our work presented in this paper falls into the category of automatic specification-based testing, we only focus the review and discussion of related work on automated specification-based testing techniques in this section. Automatic test case generation methods have been proposed for various kinds of formal specification techniques, such as algebraic specification, finite state machine, and pre-post style model-based (sometimes called state-based) specification, and different generation strategies are proposed for different formalisms. This section presents the related studies on testing based on these three kinds of formal specifications. Our description does not provide a complete coverage; only a sampling of the most relevant work directly referenced during our research is included in this section.

Algebraic specifications feature definition of abstract data types in an axiomatic manner, and their correctness requires the satisfaction of the axioms by the implementation of the functions defined in the specification. A fundamental principle for testing the implementation under test based on algebraic specifications was proposed and studied by Gannon *et al.* [55], Bouge *et al.* [56], and Bernot *et al.* [18]. The principle is characterized by choosing some instantiations of the axioms and checking whether an execution of the implementation makes the terms occurring in the instantiations yield results that satisfy the corresponding axioms. Dauchy *et al.* reported an experiment on test set derivation based on an algebraic specification of a piece of critical software [57] and Kong *et al.* described an application of testing EJB components based on algebraic specifications [58]. The focus of these researches is mainly on using equations in algebraic specifications as the basis of test oracle to check whether the implementation is satisfactory or not, but the problem of how test cases should be generated based on specifications are not well studied. Our work in this paper addresses this problem by providing a functional scenario-based vibration test case generation method, test generation criteria, and a set of new algorithms for automatic test case generation.

Finite state machine (FSM) is a mathematical model describing transitions from state to state and its extended forms including abstract state machines and statecharts are often used to model reactive behaviors of communication protocols and embedded systems [59]. Lee presented a survey on various principles and methods of testing reactive systems based on FSM, in which the conformance testing is in particular related to our interest in this paper [60]. The essential idea is that the system under test is specified by a FSM and implemented as a "black box" from which we can only observe its input/output behavior. The test sequences are generated based on the FSM and used to determine whether the implementation is consistent with the transition sequence required by the test sequence. Several specific techniques for test case generation have been proposed and developed based on the concept of "testing tree" [61], [62] or UML sequence and state machine models [22], [63]. Veanes *et al.* reported a tool known as *Spec Explorer* deployed by Microsoft since 2004 for testing reactive, object-oriented software systems [64] and Tretmans *et al.* developed a test tool known as TorX to support automatic test generation, test execution, and test analysis based on labelled transition models for reactive systems [65], [66]. The studies on FSM-based test case generation mentioned above are characterized by generating test sequences to cover all of the desired states and transitions between states. Although both FSM-based testing method and our method use formalism as the foundation for test case generation, the nature and structure of FSM are different from pre-post style specifications. Therefore, the techniques for generating test cases in the two kinds of methods are also different. The former chooses test sequences by trying to covering the desired states and transitions, while our method selects test cases based on logical expressions. In general, the FSM model is suitable for testing reactive systems, while our method is suitable for testing information systems in which rich data types are used.

A pre-post style model-based specification describes the behavior of an operation by defining the initial states before operation using a pre-condition and final states after operation using a post-condition, and various researches on software testing based on such a specification have been reported in the literature. Dick and Faivre proposed an approach to generating test data based on partitioning the conjunction of pre-condition, post-condition, and invariant for a VDM operation of interest into disjoint sub-relations by means of Disjunctive Normal Form (DNF) [2]. Such sub-relations are used to generate test data for testing both individual operations and their integrations. It does not seem to be clear from the paper how logical expressions involving only input variables (including the initial state variables) and logical expressions involving output variables (including final state variables) are automatically separated, which is necessary for automation of test data generation. The paper does not discuss the impact of the test data generated from the specification on the coverage in the program either, which is shared by other similar research as well. The similar principles are applied by Legeard *et al.* for test data generation from B or Z notation [67], [68], [69], and has been adapted in many test data generation tools, some of which use interactive theorem prover [70], [71] and others are fully automated with constraint-logic programming [72] or with heuristic algorithms driven by the syntactical form [73]. Satpathy and Butler *et al.* developed an automatic testing

approach from B formal specifications [14]. Test data are generated by performing symbolic execution over a B specification, and a test driver, which is a Java program, is obtained automatically from the test data. When it is run in conjunction with the implementation, testing is performed automatically. The main contribution of the work can be characterized as mechanical derivation of a test driver from a B specification for automatic testing, which can even handle non-determinism. TestEra [74] accepts representation constraints for such data structures and generates non-isomorphic test data by using a solution enumeration technique to use propositional constraint solver or SAT engine [75]. It generates test data for concrete representation of data structures and thus improves the test quality to cover program paths sensitive to the shape of data structures. Aichernig and Salas took the mutation testing view to propose a fault-based approach to test data generation for pre- and post-condition specifications in OCL [76], [77]. The essential idea is first to mutate the pre- and post-conditions and then try to generate test data from the specification that help find the anticipated errors. Such test data are believed to possess potential capability of finding real errors. Aichernig also investigated the issue of how VDM operation specifications can be used to derive test oracles [78]. Bouquet *et al.* extended the idea of test generation based on formal specifications to Java Modeling Language (JML), an assertion language for Java, for testing of object-oriented programs [79]. Their major contribution is the proposal of model coverage for selecting tests involving structural coverage of the specification and data coverage using a boundary analysis for numerical data. The proposed approach uses the specification both as an oracle and as a support for generating test data. Development techniques that apply a similar approach to create a test suite for safety-critical Java using JML are explored in the work [80] by Ravn and Sondergaard.

Our method presented in this paper also adopts a syntax-driven algorithmic approach to transforming a specification to a DNF, but has three distinct characteristics compared to the existing work mentioned above. First, the specification is initially not only converted into a DNF, but further into a *functional scenario form* of the specification as a more suitable format for test case generation. Second, our approach allows test cases and test oracle to be generated and analyzed from individual functional scenarios, thus achieving the effect of "divide and conquer" that can facilitate the user to easily observe the effect of testing and save the cost for result analysis. Finally, our method not only deals with the coverage of required functions in the specification, but also the coverage of execution paths in the program.

6 Conclusion

We have developed and presented a new and systematic functional scenario-based vibration method called V-Method for automatic test case generation from formal specifications. The method is suitable for testing information systems in which rich data types are used. It includes two criteria for test case generation, a group of test case generation algorithms, a V-Step for generating test cases from atomic predicates with the aim of achieving a high path coverage, and the definition of test oracle for test result analysis. We have also conducted a controlled experiment to evaluate the effectiveness of the V-Method by comparing it with the functionality-based input domain partition method (FBIDPM). The result shows that the V-Method is generally more effective in detecting bugs and gaining program path coverage than the FBIDPM. In spite of the important progress we have made in the V-Method, some challenges still remain for future work, as we have discussed in Section 4.5.

Although our method for generating test cases is discussed on the basis of formal specification in this paper, it can be used flexibly in practice, depending on specific situations. If formal notation such as SOFL or VDM is adopted in a realistic software project, the application of our method will be straightforward. However, our method can also be applied to the projects where only informal or semi-formal specifications are employed. In this case, there are two ways to apply our method. One is to apply the underlying principles and criteria of our method in both test case generation and test result analysis based on the informal or semi-formal specifications, but this application would have to be done manually. Another way is first to formalize the informal or semi-formal specification by the tester or relevant developer (if possible), and then apply our method. In this case, the tester is required to be an experienced user of formal notation. In fact, some consulting companies have specialized in formal methods, such as FormalTech Co., Ltd. [81] in Japan, and these companies can serve as a tester to apply our method in practice.

7 Future Work

Our future work will concentrate on addressing the challenges described above. In particular, we will focus our effort on improving the efficiency of our test case generation method in achieving path coverage with minimum number of test cases. To this end, we will study how the principles for defining branch distance in search-based testing [82], [83] can be properly utilized to enhance the effectiveness of the distance definition for test case generation in our V-Method and investigate how the well-established refinement calculus and functional testing theories [32], [84] can be used to establish a definitive relation between test case generation from functional scenarios in the specification and the execution of paths in the program. We will also continue to work on the construction of a supporting tool for our method on the basis of the prototype tools developed so far in our lab, which include the prototypes for automatic test case generation, automatic tracking of traversed program paths, automatic test driver construction, and automatic test result analysis. These prototypes are still simple and not well integrated together, but our experience in developing them will allow us to improve the current tools to a more efficient and integrated software tool in the future. With the completed tool, we will be interested in carrying out more experiments on testing industrial scale software.

Acknowledgments

We thank our graduate students CenCen Li, Mo Li, Takumi Amitani, Kenta Sugai, Qin Xu, and Rong Wang for their great contributions in conducting the experiment on

our testing method, and Hayato Ikeda, Yusuke Morizumi, Masahiro Yoshida, Eriko Maeyama, Kai Toda, Ye Yan, and Weihang Zhang for their contributions to the construction of various prototype software tools to support our method. Although the prototypes are not well integrated yet to make it ready to report in this paper, their work has provided us with useful feedback for improving our testing method. We would also like to thank various anonymous referees for their comments on the early drafts and sections of this paper. Finally, this work was supported in part by JSPS KAKENHI Grant Number 26240008, the NII Collaborative Research Program, and the SCAT Foundation.

REFERENCES

[1] A. Abran and J. W. M. Eds., *Guide to the Software Engineering Body of Knowledge*. Washington, DC, USA: IEEE Comput. Soc., 2004.
[2] J. Dick and A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications," in *Proc. Int. Symp. Formal Methods Europe*, 1993, pp. 268–284.
[3] P. Stocks and D. Carrington, "A framework for specification-based testing," *IEEE Trans. Softw. Eng.*, vol. 22, no. 11, pp. 777–793, Nov. 1996.
[4] S. J. Galler and B. K. Aichernig, "Survey on test data generation tools: An evaluation of white- and gray-box testing tools for C#, C++, Eiffel, and Java," *Int. J. Softw. Tools Technol. Transfer*, vol. 16, pp. 727–751, 2014.
[5] C. Hoare, "An axiomatic basis of computer programming," *Commun. ACM*, vol. 12, pp. 576–580, 1969.
[6] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 455–495, Jul. 1982.
[7] E. M. Clarke and E. A. Emerson, "Synthesis of synchronization skeletons for branching time temporal logic," in *Proc. Workshop Logic Programs*, 1981, pp. 52–71.
[8] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 21:1–21:54, 2009.
[9] G. T. Leavens, K. Rustand, M. Leino, and P. Muller, "Specification and verification challenges for sequential object-oriented programs," *Formal Aspects Comput.*, vol. 19, pp. 159–189, 2007.
[10] D. L. Parnas, "Really rethinking formal methods," *Computer*, vol. 43, no. 1, pp. 28–34, Jan. 2010.
[11] S. Liu, Y. Chen, F. Nagoay, and J. McDermid, "Formal specification-based inspection for verification of programs," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1100–1122, Sep. 2012.
[12] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. New York, NY, USA: McGraw-Hill, 2001, ch. 14.
[13] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.
[14] M. Satpathy, M. Butler, M. Leuschel, and S. Ramesh, "Automatic testing from formal specifications," in *Proc. Int. Conf. Tests Proofs*, 2007, pp. 95–113.
[15] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Inf. Softw. Technol.*, vol. 43, pp. 841–854, 2001.
[16] C. Cadar *et al.*, "Symbolic execution for software testing in practice - Preliminary assessment," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 1066–1071.
[17] P. G. Larsen, J. Fitzgerald, and T. Brookes, "Applying formal specification in industry," *IEEE Softw.*, vol. 13, no. 3, pp. 48–56, May 1996.
[18] G. Bernot, M. C. Gaudel, and B. Marre, "Software testing based on formal specifications: A theory and a tool," *Softw. Eng. J.*, vol. 6, no. 6, pp. 387–405, 1991.
[19] J. McDonald and P. Strooper, "Translating object-Z specifications to passive test oracles," in *Proc. 2nd Int. Conf. Formal Eng. Methods*, 1998, pp. 165–174.
[20] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 1–39, 2009.
[21] S. Liu, T. Hayashi, K. Takahashi, K. Kimura, T. Nakayama, and S. Nakajima, "Automatic transformation from formal specifications to functional scenario forms for automatic test case generation," in *Proc. 9th Int. Conf. Softw. Methodologies Tools Techn.*, 2010, pp. 383–397.
[22] J. Offutt and A. Abdurazik, "Generating test cases from UML specifications," in *Proc. 2nd Int. Conf. Unified Model. Lang.*, 1999, pp. 416–429.
[23] S. Liu and S. Nakajima, "A decompositional approach to automatic test case generation based on formal specifications," in *Proc. 4th IEEE Int. Conf. Secure Softw. Integr. Rel. Improvement*, 2010, pp. 147–155.
[24] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann, "Automated test suite generation for time-continuous Simulink models," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 595–606.
[25] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann, "Test generation and test prioritization for Simulink models with dynamic behavior," *IEEE Trans. Softw. Eng.*, vol. 45, no. 9, pp. 919–944, Sep. 2019, doi: 10.1109/TSE.2018.2811489.
[26] P. Zhao and S. Liu, "A software tool to support the "Vibration" method," in *Proc. 7th Int. Workshop SOFL+MSVL*, 2017, pp. 171–186.
[27] S. Liu, F. Nagoya, Y. Chen, M. Goya, and J. A. McDermid, "An automated approach to specification-based program inspection," in *Proc. 7th Int. Conf. Formal Eng. Methods*, 2005, pp. 421–434.
[28] S. Liu, *Formal Engineering for Industrial Software Development Using the SOFL Method*. Berlin, Germany: Springer-Verlag, 2004, isbn: 3–540-20602-7.
[29] IFAD, *VDMTOOLS for Quality Software on Schedule - VDM-SL Toolbox User Manual, The IFAD VDM-SL Language*, The Institute of Applied Computer Science, 1999.
[30] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Commun. ACM*, vol. 31, no. 6, pp. 676–686, Jun. 1988.
[31] P. Ammann and J. A. Offutt, *Introduction to Software Testing*. Cambridge, U.K.: Cambridge Univ. Press, 2008.
[32] C. Morgan, *Programming From Specifications*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 1994.
[33] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *Proc. 28th Annu. Int. Symp. Fault-Tolerant Comput.*, 1998, pp. 230–239.
[34] W. Zhang and S. Liu, "Supporting tool for automatic specification-based test case generation," in *Proc. 2nd Int. Workshop Structured Object-Oriented Formal Lang. Method*, 2012, pp. 12–25.
[35] J. Dawes, *The VDM-SL Reference Guide*. New York, NY, USA: Pitman, 1991.
[36] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*. Englewood Cliffs, NJ, USA: Prentice Hall, 1996.
[37] J. R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: An open toolset for modelling and reasoning in event-B," *Int. J. Softw. Tools Technol. Transfer*, vol. 12, no. 6, pp. 447–466, 2010.
[38] S. Liu, "Testing-based formal verification for theorems and its application in software specification verification," in *Proc. 10th Int. Conf. Tests Proofs*, 2016, pp. 112–129.
[39] N. Manthey, "Solver description of RISS 2.0 and PRISS 2.0," Knowledge Representation and Reasoning, Technical University Dresden, KRR Teport 12–02, 2012.
[40] B. Dutertre, "Yices 2.2," in *Proc. Int. Conf. Comput.-Aided Verification*, 2014, pp. 737–744.
[41] L. D. Moura and N. Bjorner, "Z3: An efficient SMT solver," in *Proc. 11th Eur. Join Conf. Theory Practice Softw. 14th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.
[42] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," in *Proc. 3rd Int. Conf. Qual. Softw.*, 2003, pp. 4–11.
[43] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Object distance and its application to adaptive random testing of object-oriented programs," in *Proc. 1st Int. Workshop Random Testing*, 2006, pp. 55–63.
[44] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: Adaptive random testing for object-oriented software," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 71–80.
[45] D. I. K. Sjoberg *et al.*, "A survey of controlled experiments in software engineering," *IEEE Trans. Softw. Eng.*, vol. 31, no. 9, pp. 733–753, Sep. 2005.
[46] B. Beizer, *Black-Box Testing*. Hoboken, NJ, USA: Wiley, 1995.
[47] L. Lazic and N. Mastorakis, "Cost effective software test metrics," *WSEAS Trans. Comput.*, vol. 7, no. 6, pp. 599–619, Jun. 2008.

[48] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research - An initial survey," in *Proc. 22nd Int. Conf. Softw. Eng. Knowl. Eng.*, 2010, pp. 374–379.
[49] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Comput.*, vol. 11, no. 4, pp. 34–41, Apr. 1978.
[50] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proc. Int. Symp. Softw. Testing Anal.*, 1993, pp. 139–148.
[51] Q. V. Nguyen and L. Madeyski, "Problems of mutation testing and higher order mutation testing," in *Proc. Int. Conf. Intell. Syst. Comput.*, 2014, pp. 157–172.
[52] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial complier equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 936–946.
[53] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "Treats to the validity of mutation-based test assessment," in *Proc. 25th Int. Symp. Testing Anal.*, 2016, pp. 354–365.
[54] M. Harman and S. Danicic, "The relationship between program dependence and mutantion analysis," in *Mutation Testing for the New Century*, W. E. Wong, Ed. New York, NY, USA: Springer, 2001, pp. 5–14.
[55] J. Gannon, P. McMullin, and R. Hamlet, "Data abstraction implementation, specification and testing," *ACM Trans. Program. Lang. Syst.*, vol. 3, no. 3, pp. 211–223, 1981.
[56] L. Bouge, N. Choquet, L. Fribourg, and M.-C. Gaudel, "Test set generation from Algebraci specifications using logic programing," *J. Syst. Softw.*, vol. 6, no. 4, pp. 343–360, 1986.
[57] P. Dauchy, M.-C. Gaudel, and B. Marre, "Using algebraic specifications in software testing: A case study on the software an automatic subway," *J. Syst. Softw.*, vol. 21, no. 3, pp. 229–244, Jun. 1993.
[58] L. Kong, H. Zhu, and B. Zhou, "Automated testing EJB components based on algebraic specifications," in *Proc. 31st Annu. Int. Comput. Softw. Appl. Conf.*, 2007, pp. 717–722.
[59] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-Based Testing of Reactive Systems*. Berlin, Germany: Springer-Verlag, 2005.
[60] D. Lee, "Principles and methods of testing finite state machines - A survey," *Proc. IEEE*, vol. 84, no. 8, pp. 1090–1123, Apr. 1996.
[61] T. Chow, "Testing software designs modeled by finite-state machines," *IEEE Trans. Softw. Eng.*, vol. 4, no. 3, pp. 178–187, May 1978.
[62] S. Fujiwara, G. V. Bochmann, F. Khendek, and M. Amalou, "Test selection based on finite state models," *IEEE Trans. Softw. Eng.*, vol. 17, no. 6, pp. 591–603, Jun. 1991.
[63] A. Bandyopadhyay and S. Ghosh, "Test input generation using UML sequence and state machines models," in *Proc. 2nd Int. Conf. Softw. Testing Verification Validation*, 2009, pp. 121–130.
[64] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reacitve systems with spec explorer," in *Formal Methods and Testing*, R. Hierons, J. Bowen, and M. Harman, Eds. Berlin, Germany: Springer-Verlag, 2008, pp. 39–76.
[65] G. J. Tretmans and H. Brinksma, "TorX: Automated model-based testing," in *Proc. 1st Eur. Conf. Model-Driven Softw. Eng.*, 2003, pp. 31–43.
[66] J. Schmaltz and J. Tretmans, "On conformance testing for timed systems," in *Proc. Int. Conf. Formal Modelling Anal. Timed Syst.*, 2008, pp. 250–264.
[67] B. Legeard, F. Peureux, and M. Utting, "Automatic boundary testing from Z and B," in *Proc. Int. Symp. Formal Methods Europe*, 2002, pp. 21–40.
[68] B. Legeard, F. Peureux, and M. Utting, "Controlling test case explosion in test generation from B formal models," *Softw. Testing Verification Rel.*, vol. 14, pp. 81–103, 2004.
[69] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Mateo, CA, USA: Morgan Kaufmann, 2007.
[70] S. Helke, T. Neustupny, and T. Santen, "Automating test case generation from Z specifications with Isabelle," in *Proc. 10th Int. Conf. Z Users*, 1997, pp. 52–71.
[71] S. Burton, "Automated testing from Z specifications," Univ. York, York, U.K., Tech. Rep. YCS-2000–329, 2000.
[72] L. Lucio and M. Samer, "Technology of test-case generation," in *Model-Based Testing of Reactive Systems*, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds. Berlin, Germany: Springer-Verlag, 2005, pp. 323–454.
[73] M. Donat, "Automating formal specification-based testing," in *Proc. Colloq. Trees Algebra Program.*, 1997, pp. 833–848.
[74] S. Khurshid and D. Marinov, "TestEra: Specification-based testing of Java programs using SAT," *Automated Softw. Eng.*, vol. 11, no. 4, pp. 403–434, 2004.
[75] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson, "A case for efficient solution enumeration," in *Proc. Int. Conf. Theory Appl. Satisfiability Testing*, 2003, pp. 297–298.
[76] B. K. Aichernig and P. A. P. Salas, "Test case generation by OCL mutation and constraint solving," in *Proc. 5th Int. Conf. Qual. Softw.*, 2005, pp. 64–71.
[77] B. K. Aichernig, "Model-based mutation testing of reactive systems - From semantics to automated test case generation," in *Theories of Programming and Formal Methods*. Berlin, Germany: Springer, 2013, pp. 23–36.
[78] B. K. Aichernig, "Automated black-box testing with abstract VDM Oracles," in *Proc. 18th Int. Conf. Comput. Safety Rel. Secur.*, 1999, pp. 250–259.
[79] F. Bouquet, F. Dadeau, and B. Legeard, "Automated boundary test generation from JML specifications," in *Proc. Int. Symp. Formal Methods*, 2006, pp. 428–443.
[80] A. P. Ravn and H. Sondergaard, "A test suite for safety-critical Java using JML," in *Proc. 11th Int. Workshop Java Technol. Real-Time Embedded Syst.*, 2013, pp. 80–88.
[81] Formaltech Co., Ltd. [Online]. Available: http://www.formaltech.co.jp/
[82] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in *Proc. Int. Conf. Automated Softw. Eng.*, 1998, pp. 285–288.
[83] P. McMinn, "Search-based software test data generation: A survey," *J. Softw. Testing Verification Rel.*, vol. 14, no. 2, pp. 105–156, 2004.
[84] J. Goodenough and S. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Softw. Eng.*, vol. 1, no. 2, pp. 156–173, Jun. 1975.

Shaoying Liu (Fellow, IEEE) received the PhD degree in computer science from the University of Manchester, Manchester, United Kingdom, in 1992. He is currently a professor of software engineering with the Hiroshima University, Japan. Previously, he was a professor with the Hosei University from April 2000 to March 2020. 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 more than 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 of the *IEEE Transactions on Reliability* and was on the Editorial Board of the *Journal of Software Testing, Verification and Reliability* (STVR). He is also the BCS fellow and a member of the JSSST and the IPSJ.

Shin Nakajima received the BS and MS degrees from the University of Tokyo, Tokyo, Japan, in 1979 and 1981, respectively, and the PhD degree, in 2000. He is currently a professor with the National Institute of Informatics, Tokyo, Japan. His current interests include formal methods, automated verification, and software testing. He is a member of the JSSST and IPSJ.