Verifying Consistency and Validity of Formal Specifications by Testing[⋆]

Shaoying Liu

Faculty of Information Sciences
Hiroshima City University, Japan
shaoying@cs.hiroshima-cu.ac.jp
http://www.sel.cs.hiroshima-cu.ac.jp/~liu/

Abstract. Detecting faults in specifications can help reduce the cost and risk of software development because incorrect implementation can be prevented early. This goal can be achieved by verifying the consistency and validity of specifications. In this paper we put forward *specification testing* as a practical technique for verification and validation of formal specifications. Our approach is to derive *proof obligations* from a specification and then test them, in order to detect faults leading to the violation of consistency or validity of the specification. We describe proof obligations for various consistency properties of a specification, and suggest the use of five strategies for testing them. We provide a method for testing implicit specifications by evaluation rather than by prototyping, and criteria for interpreting the meaning of test results.

1 Introduction

It is desirable and important to detect faults in a specification (for requirements or design) before it is implemented, in order to reduce the cost and risk of software development [1, 2]. Faults may arise if the consistency of the specification is violated or the user requirements are misrepresented by the specification. A specification is consistent if there exists a computational model for its implementation. A specification is valid if it expresses the user requirements satisfactorily.

To support verification and validation of formal specifications, we suggest *specification testing* as a practical technique. Our target is to deal with implicit or nonprocedural specifications, possibly involving pre and postconditions like those in VDM [3] or Z [4]. Such a specification is often not executable, but can often be evaluated for a given input and output. In addition, we also expect to deal with the complex constructs composed of implicit formal specifications for components. Our approach is to combine the ideas of formal proof and program

[⋆] This work is supported in part by the Ministry of Education of Japan under Grant-in-Aid for Scientific Research on Priority Areas (A) (No. 09245105), Grant-in-Aid for Scientific Research (B) (No. 11694173), and Grant-in-Aid for Scientific Research (C) (No. 11680368) testing. The principle of this combination is that formal proof obligations indicate what to verify for what purpose, whereas testing offers the idea of using test cases to check proof obligations.

1.1 Specification Testing

By specification testing, we mean presentation of inputs and outputs to a specification, and evaluation to obtain a result—usually a truth value. As the postcondition of an implicit operation usually describes the relation between its inputs and outputs, an evaluation of the postcondition needs both inputs and outputs. This is slightly different from program testing, as discussed in detail later in this section. Our concrete approach to testing is to derive a proof obligation expressing the consistency property of the *testing target* (e.g., invariant, operation, composition of operations), and then test the proof obligation with test cases—selected inputs and outputs. The proof obligation is a necessary prerequisite for the testing target to be consistent in terms of the semantics of the formal specification language in which the target is written. For this reason, the proof obligation is usually derived based on the semantics of the testing target.

Similar to conventional program testing [5], testing a specification also consists of three steps: (1) test case generation; (2) evaluation of proof obligations that are logical expressions derived from the specification; and (3) analysis of test results, as illustrated in Figure 1. Two methods for generating test cases can be used. One is to produce test cases based on the proof obligations derived from the specification. This is similar to *structural testing* for programs where test cases are based on examination of program structure. In this method, there is no need to provide expected test results, because the meaning of the testing is interpreted based on the established criteria (which is mainly for checking consistency). Another method is based on informal user requirements. This is similar to *functional testing* for programs, where test cases are based on a functional description of the program. In this method, expected test results are required, in order to check whether or not the specification expresses satisfactorily the user requirements. An evaluation of a logical expression is a process of computing the expression with a test case to yield **true** or **false**. Analysis of test results determines the nature of the test, and possibly indicates the existence of faults in the specification.

To test an entire specification, *unit testing* and *integration testing* can be conducted for different objectives. Unit testing aims to detect faults in each component which can be an invariant, operation or object; whereas integration testing tries to uncover faults occurring in the integration of operations (e.g., functional compositions by control constructs or message communications), and to check whether the required services are specified satisfactorily.

When testing an operation (which can be a method for object-oriented specification language like Object-Z [6]), it is necessary to treat the state variables before and after the operation, for example, $\overleftarrow{x}$ and x in VDM [3]; x and x' in Z [4], as

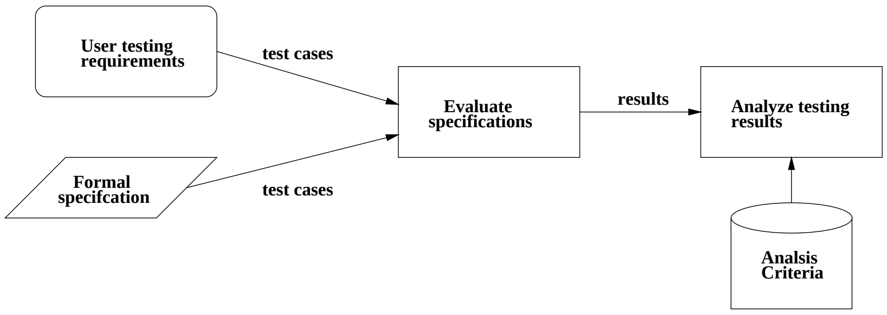

Fig. 1. The process of specification testing

inputs and outputs of the operation, respectively. This will allow an evaluation of the postcondition of the operation and a verification of whether the change of the state by this operation is satisfactory in both its consistency and validity.

Specification testing has two major differences from program testing. To test the specification of an operation, for example, we need test cases for both input and output variables, whereas we need only for input variables to run the program in program testing (although we need to supply the expected results for test results analysis). Another difference is that testing a specification involves evaluation of logical expressions but does not involve running any program, which allows the system to be tested before its implementation, whereas testing a program entails running the program.

1.2 Related Work

Specification testing is much less researched than program testing, and only a few reports are in the literature. Testing implicit formal specifications is especially not well studied. In contrast to this situation, *specification-based testing* for programs has attracted much more attention from the research community.

An early and important work on testing formal specifications was done by Kemmerer [2]. He argued that testing specifications early in the software life cycle to discover whether a formal specification has a *satisfiable* implementation and whether the specification satisfies its critical and desired functional requirements can help ensure the reliability of systems and reduce cost. Kemmerer proposed two approaches to testing nonprocedural formal specifications, and applied them to specifications written in the *Ina Jo* language. One is *by prototyping*, that is, transforming a nonprocedural specification into a procedural form and then using the latter as a rapid prototype for testing. The other approach is *by symbolic execution*, that is, performing a symbolic execution of the sequence of operations and checking the resultant symbolic values to see if they define the desired set of resultant states.

Recent work on testing object-oriented formal specifications was undertaken by Chen and Liu [7]. We suggested using specification testing as an alternative to *theorem proving* in order to verify whether the required properties are satisfied by the specification for finite test cases. However, the proposed technique was applied only to specification components (e.g., invariants, methods), not to their integration. No well-defined criteria for test case generation were given either.

Specification-based testing is another area of research related to specification testing [8, 9, 10]. Although related to each other, specification testing and specification-based testing are different in a number of ways. First, their targets are different. The former aims to test the specification itself, whereas the latter aims to test the program that implements the specification. Second, the former tries to detect faults in requirements and design, whereas the latter attempts to find faults in the program which may lead to violation of the specification. Third, test cases are not necessarily generated based on the structure of the specification for the former, whereas for the latter test cases are generated based on the structure of the specification. Finally, testing specifications involves evaluation of logical expressions, whereas specification-based testing entails running the program.

1.3 Contributions

We make three major contributions in this paper. Firstly, we suggest a new approach to testing implicit formal specification by evaluation of proof obligations derived from the specification. Secondly, we suggest the use of five strategies for testing logical expressions, which can be applied to test proof obligations for consistency and specifications themselves for validity. Finally, we provide criteria for interpreting the meaning of a test result to determine whether or not a fault is found by the test.

We apply the proposed approach to SOFL (Structured Object-based Formal Language), and describe proof obligations for verifying consistency properties of the SOFL constructs: invariants, condition processes (similar to VDM operations), condition data flow diagrams which integrate condition processes, and decompositions of condition processes.

We choose SOFL to apply the proposed technique, because SOFL integrates the advantages of classical data flow diagrams, Petri nets, and VDM-SL, and has begun to be applied to real projects [11, 12, 13]. Due to this feature, the technique applied to SOFL specifications in this paper can also be applied to commonly used data flow diagrams, Petri nets and model-oriented formal specifications, such as VDM, Z, and B-method [14]. For the sake of space, no introduction to SOFL is given in this paper. The reader who is interested in the details of SOFL can refer to the author's previous publications [11, 15, 12].

The remainder of this paper is organized as follows. Section 2 describes strategies for testing logical expressions in general which can be applied to test proof obligations. Section 3 discusses *unit testing*, including testing of specification invariants and condition processes. Section 4 focuses on *integration testing* to show how to test condition data flow diagrams. Section 5 addresses the issue of verifying decomposition of condition processes. Finally, in section 6 we give conclusions and outline future research.

2 Testing Logical Expressions

As described previously, the fundamental step in testing a specification is to test the logical expressions that represent the proof obligations derived from the specification. In this section we discuss strategies for testing logical expressions.

Definition 2.1 Let $P(x_1, x_2, \ldots, x_n)$ be a logical expression containing free variables $x_1, x_2, \ldots, x_n$. A *test case* for P is *a group of values* $v_1, v_2, \ldots, v_n$ bound to $x_1, x_2, \ldots, x_n$, respectively, and a *test set* for P is *a non-empty set of test cases.*

Note that a test set for P can be a single test case as well.

Definition 2.2 Let P be a logical expression. A *test suite* for P is a set of pairs $\{(T_c^1, E_r^1), (T_c^2, E_r^2), \ldots, (T_c^m, E_r^m)\}$ where T_c^i $(i = 1...m)$ are test cases and E_r^i are expected results corresponding to test cases T_c^i.

As testing a logical expression involves evaluation of the expression to either **true** or **false**, each E_r^i in fact is a truth value.

Definition 2.3 Let P be a logical expression and T_d a test set for P. A *test* of P is a set of evaluations of P with all the test cases in the test set T_d.

Definition 2.4 Let P be a logical expression and T_s a test suite for P. A *test report* of P can be one of the two forms. One is a set of triplets $\{(T_c^1, E_r^1, A_r^1), (T_c^2, E_r^2, A_r^2), \ldots, (T_c^m, E_r^m, A_r^m)\}$ where T_c^i $(i = 1...m)$ are test cases; E_r^i and A_r^i are expected and actual results corresponding to test cases T_c^i, respectively. Another form is a set of pairs $\{(T_c^1, A_r^1), (T_c^2, A_r^2), \ldots, (T_c^m, A_r^m)\}$.

For example, suppose $P \equiv x > 0 \land x < 10/y$, where $\equiv$ means "is defined syntactically as"; x and y denote a real number and an integer, respectively, then

(1) $(x = 2.0, y = 5)$ is a test case for P.

(2) $\{(x = -1.0, y = 4), (x = 1.0, y = 0), (x = 1.5, y = 2)\}$ is a test set for P.

(3)

x	y	E_r
-1.0	4	**false**
1.0	0	**true**
1.5	2	**true**

shows a test suite for P.

(4)

x	y	E_r	P
-1.0	4	**false**	**false**
1.0	0	**true**	*
1.5	2	**true**	**true**

shows a test report of P, where * is a logical value, representing "undefined".

For convenience, it is assumed that logical expressions under discussion are in disjunctive normal form (DNF). The primary intent is that each clause in each logical expression is tested independently.

We suggest five strategies for testing logical expressions with different objectives, each imposing a different constraint on selection of test cases.

Let $P \equiv P_1 \vee P_2 \vee \cdots \vee P_n$ be a disjunctive normal form and $P_i \equiv Q_i^1 \wedge Q_i^2 \wedge \cdots \wedge Q_i^m$ be a conjunction of relational expressions Q_i^j which are atomic components, where $i = 1...n$ and $j = 1...m$. We treat a quantified expression as an atomic logical expression (the same level as a relational expression) in P_i. When generating a test case for a universal quantifier, for example $\forall_{x \in \mathbf{nat}} \cdot x + 1 > x$, we generate a finite subset of the infinite natural number type **nat**, say $\{0, 1, ..., 1000\}$, to replace **nat** in this quantified expression. Let T_d be a test set for P.

Strategy 1 Evaluate P with T_d to **true** and **false**, respectively.

This strategy is illustrated by table 1.

Table 1. Strategy 1

P
true
false

Strategy 2 Evaluate P_i with T_d for every $i = 1...n$ to **true** and **false**, respectively.

Table 2 explains this strategy.

Table 2. Strategy 2

P_1	P_2	P_3	...	P_n
true	$\star$	$\star$	...	$\star$
false	$\star$	$\star$	...	$\star$
$\star$	**true**	$\star$	...	$\star$
$\star$	**false**	$\star$	...	$\star$
.	.	.	...	.
.	.	.	...	.
.	.	.	...	.
$\star$	$\star$	$\star$	...	**true**
$\star$	$\star$	$\star$	...	**false**

where $\star$ denotes either **true** or **false**.

Although it is possible to test the cases of each disjunct P_i being **true** and **false** respectively by using this strategy, we cannot guarantee that each disjunct be evaluated to **true** while every other disjunct to **false**. To overcome this weakness, a strengthened strategy is given as follows.

Strategy 3 Evaluate P_i with T_d to **true** while all $P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n$ to **false**, and to **false** while all $P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_n$ to **true**, respectively.

This strategy is illustrated by table 3.

However, for some logical expressions, for example, $x > 0 \vee x > 3$, it is impossible to directly apply this strategy because when $x > 3$ evaluates to **true**, there is no way to evaluate $x > 0$ to **false**. In this case, either the logical expression (or the corresponding specification) needs to be modified or **Strategy 2** needs to be applied.

Table 3. Strategy 3

P_1	P_2	P_3	...	P_n
true	**false**	**false**	...	**false**
false	**true**	**false**	...	**false**
false	**false**	**true**	...	**false**
.	.	.	...	.
.	.	.	...	.
.	.	.	...	.
false	**false**	**false**	...	**true**

Strategy 4 When evaluating P_i to **true** with T_d, evaluate every Q_i^j to **true**. When evaluating P_i to **false**, evaluate Q_i^j to **false** for every $j = 1...m$, respectively.

This strategy is for testing a disjunct of P which is a conjunction of sub-expressions. Table 4 illustrates this strategy.

Table 4. Strategy 4

Q_1	Q_2	Q_3	...	Q_m
true	**true**	**true**	...	**true**
false	$\star$	$\star$	...	$\star$
$\star$	**false**	$\star$	...	$\star$
.	.	.	...	.
.	.	.	...	.
.	.	.	...	.
$\star$	$\star$	$\star$	...	**false**

Strategy 5 When evaluating P_i to **true** with T_d, evaluate every Q_i^j to **true**. When evaluating P_i to **false**, evaluate Q_i^j to **false** while all $Q_i^1, \ldots, Q_i^{j-1}, Q_i^{j+1}$, ..., Q_i^n to **true**.

This strategy imposes a stronger restriction on evaluation of conjuncts than previous one. It is illustrated by table 5.

By convention in program testing, a *successful test* means that a fault is detected by the test, whereas a *failed test* means that no fault is detected by the test. We also follow this convention in this paper.

Table 5. Strategy 5

Q_1	Q_2	Q_3	...	Q_n
true	**true**	**true**	...	**true**
false	**true**	**true**	...	**true**
true	**false**	**true**	...	**true**
true	**true**	**false**	...	**true**
.	.	.	...	.
.	.	.	...	.
.	.	.	...	.
true	**true**	**true**	...	**false**

3 Unit Testing

Unit testing includes testing of invariants and condition processes.

3.1 Testing Invariants

For brevity, our discussion focuses on invariants which involve only a single bound variable. The same testing method can be easily extended to invariants containing multiple bound variables.

Let an invariant Inv be

forall[x **inset** D | $P(x)$]

where D can be a basic type (e.g., integers, natural numbers) or a constructed type. For the meaning of SOFL operators occurring in this paper, see Appendix A.

The invariant describes a property of all the elements of the type D which is expected to be sustained throughout the entire system. To enable the invariant to serve this purpose, it is necessary to ensure that the invariant is defined consistently and as desired. This can be tested by applying any of the **Strategy 1** to **3**.

Consistency Testing Consistency testing aims to check whether an invariant satisfies the required proof obligation or not. To this end, we need to give a precise meaning of the proof obligation for an invariant.

Definition 3.1 Let $D = R$ (D is a type identifier declared as the type R) and inv = **forall**[x **inset** D | $P(x)$] be an invariant of D (actually both the declaration and the invariant define the type D). The proof obligation for ensuring its consistency is:

$$\exists_{r \in R} \cdot P(x).$$

In other words, as long as we find a value of the type R that satisfies the property P, we will have demonstrated that the invariant inv is consistent. However, if none of the test cases in a test set of such kind satisfies the property P, we cannot assert that the invariant is inconsistent because of the limitation of test set, although it may weaken the belief that the invariant is consistent.

For example, let the type Customer be defined as

```
Customer = composed of
           id: int
           name: string
           end.
```

where **int** denotes integers.

Its invariant Inv be

forall[x **inset** Customer | x.id >= 0010 **and** x.id < 1000 **and** **len**(x.name) <= 15].

Testing this invariant can be done by testing its proof obligation:

$$\exists_{r \in R} \cdot \text{r.id >= 0010 } \textbf{and} \text{ r.id < 1000 } \textbf{and} \text{ } \textbf{len}\text{(r.name) <= 15},$$

where

```
R ≡ composed of
    id: int
    name: string
    end.
```

To this end, we need to generate test cases for the bound variable r in the proof obligation that are values of the type R.

A test report containing five test cases generated by applying **Strategy 1** is given in table 6.

Table 6. A test report for the proof obligation

r	Proof obligation
(0000, “Mark”)	**false**
(0001, “John”)	**false**
(0011, “Mike”)	**true**
(0350, “Chris”)	**true**
(0023, “Ginny”)	**true**

Obviously this test is a failed test because the property given in the proof obligation evaluates to **true** on three test cases (although only one is sufficient for our purpose). By definition this invariant is consistent.

Validity Testing Validity testing of an invariant is intended to verify the property defined by the invariant against the user or designer requirements. This can be done by evaluating directly the invariant itself rather than its proof obligation. When generating a test set, the expected test results must be supplied for test result analysis, that is, a complete test suite must be produced. The test suite may be generated on the basis of informal user requirements, it may also be produced based on the structure of the invariant using any of **Strategy 1** to **3** given in section 2.

Consider the same invariant *Inv* as an example. A test report containing five test cases is given in table 7.

Table 7. A test report for invariant *Inv*

x	Expected results	*Inv*
(0000, “Mark”)	**false**	**false**
(0001, “John”)	**false**	**false**
(1999, “Sue”)	**true**	**false**
(0011, “Mike”)	**true**	**true**
(0350, “Chris”)	**true**	**true**

As the expected test result for the third test case (i.e., **true**) is different from the actual test result **false**, this test is a **successful test** by definition, indicating the existence of a fault in the invariant (assuming there is no error in the expected result). The fault might be x.id < 1000 as the intended invariant may be

forall[x **inset** Customer | x.id >= 0010 **and** x.id < 10000 **and** **len**(x.name) <= 15].

3.2 Testing Condition Processes

A condition process in SOFL is like an operation in VDM [3]. To test a condition process, we need to generate test cases for the input parameters, output parameters, and external variables before and after a *firing* (execution) of the condition process. Similar to testing an invariant, two aspects of a condition process can be tested, *consistency* and *validity.*

Consistency The goal of consistency testing is to check whether a condition process satisfies its proof obligation using test cases. For brevity, our discussion is limited to the condition processes that involve only one external variable.

Let CP be condition process, represented concisely as

$CP : [I,\ O,\ E,\ pre,\ post]$

where I is the set of input parameters; O is the set of output parameters; E is the set of external variables occurring in *pre* and/or *post* (e.g., data stores connecting to this condition process can be such variables); *pre* and *post* are the pre and postconditions of condition process CP.

Note that in the postcondition, we use $\tilde{}x$ and x, for example, to represent the values of external variable x before and after a firing of the condition process, respectively.

The proof obligation for condition process CP is

$$\forall_{\tilde{}x \in \Sigma} \bullet pre(I,\ \tilde{}x) \Rightarrow \exists_{x \in \Sigma} \bullet post(I,\ x,\ \tilde{}x,\ O) \tag{1}$$

It means that for every initial value of the external variable $\tilde{}x$ in the state Σ, if it, together with all the values bound to input parameters, satisfies the precondition, there must exist a final value for the external variable x such that it, together with its initial value and all the values bound to input and output parameters, satisfies the postcondition. For brevity, we call all the values bound to input parameters, *read only* state variables, and decorated external variables (e.g. $\tilde{}x$) *inputs* and all the values bound to output parameters and *write and read* state variables *outputs.*

For convenience, we represent a (single) test case as a pair of sets $< I_v, O_v >$, where I_v is input and O_v output. For example, $< \{2, 3, 4\},\ \{20, 32\} >$ can be a test case for a condition process with three input parameters and two output parameters.

Definition 3.2 Let $T_d = \{T_1, T_2, \ldots, T_n\}$ be a test set and $T_i =< I_v^i, O_v^i >$ for $i = 1...n$. If for every I_v^i satisfying the precondition there exists an O_v^j $(j = 1...n)$ in T_d such that the proof obligation 1 evaluates to **true**, we say the test with T_d is a failed test for CP. Otherwise, the test is a non-confident test.

It is worth noting that in evaluating the pre and postconditions it must take into account that the values of all the variables satisfy the invariants of their associated types. In other words, by saying that the values of all the variables satisfy the pre or postcondition we mean that they satisfy both the pre or postcondition and the invariants of their associated types.

If the proof obligation is discharged by the test, namely the proof obligation evaluates to the expected truth values on the test set used for the test, the test will be a failed test because no fault is detected by this test. A non-confident test indicates a doubt whether the testing target (e.g., condition process) satisfies the required proof obligation. However, this does not necessarily imply that a fault exists in the target, as the test set, in particular the values bound to output variables, may not be selected appropriately.

For example, a condition process Search is given below. It takes an integer greater than or equal to five and checks whether it occurs in a given sequence. If the integer is found, its position in the sequence, a natural number, is supplied as the result; otherwise, zero is provided as the result.

```
c-process Search(x: int) index: nat
 ext  rd list: seq of int
 pre  x >= 5 and elems(list) <> { }
 post exists[i inset inds(list) | list[i] = x and index = i] or
        x notin elems(list) and index = 0
end-process
```

To test this condition process, we need to produce a set of test cases for the input parameter x, the external variable list, and the output parameter index. By taking **Strategy 1**, we produce a test report given in table 8.

Table 8. A test report for Search

x	list	index	prd1	prd2	pod1	pod2	prec	postc	spec
5	[0, 8]	0	**true**	**true**	**false**	**true**	**true**	**true**	**true**
5	[0, 8]	2	**true**	**true**	**false**	**false**	**true**	**false**	**false**
15	[ ]	2	**true**	**false**	**false**	**false**	**false**	**false**	**true**
6	[0, 2, 6]	3	**true**	**true**	**true**	**false**	**true**	**true**	**true**

where

prd1 ≡ x >= 5,
prd2 ≡ **elems**(list) <> { },
prec ≡ x >= 5 **and elems**(list) <> { },
pod1 ≡ **exists**[i **inset inds**(list) | list[i] = x **and** index = i],
pod2 ≡ x **notin elems**(list) **and** index = 0,

postc ≡ **exists**[i **inset inds**(list) | list[i] = x **and** index = i] **or**
x **notin elems**(list) **and** index = 0,
spec ≡ prec ⇒ postc.

In this test the two groups of inputs (5, [0, 8]) and (6, [0, 2, 6]) satisfy the precondition prec, respectively, and there exist the outputs 0 and 3 such that the proof obligation spec evaluates to **true**, respectively. By definition this is a failed test for Search. Note that we do not consider the input (15, [ ]) as it does not satify the precondition, nor the test case ((5, [0, 8]), 2) because of the existence of test case ((5, [0, 8]), 0).

It is evident that the more quality tests are conducted, the more evidence they provide as to the satisfiability of the condition process.

Validity Validity testing for a condition process aims at verifying the consistency between the specified functionality of the condition process and the desired functionality required by either the user or designer.

When carrying out a validity testing, we need to produce both test cases and the expected results. If the actual test results are the same as the expected results, the test is a failed test, namely no fault is detected by the test. Otherwise, it is a successful test.

Table 9 shows a failed test for condition process Search. E_r represents the expected results.

Table 9. A validation test for Search

x	list	index	E_r	prd1	prd2	pod1	pod2	prec	postc	spec
5	[0, 8]	0	**true**	**true**	**true**	**false**	**true**	**true**	**true**	**true**
5	[0, 8]	2	**false**	**true**	**true**	**false**	**false**	**true**	**false**	**false**
15	[ ]	2	**true**	**true**	**false**	**false**	**false**	**false**	**false**	**true**
6	[0, 2, 6]	3	**true**	**true**	**true**	**true**	**false**	**true**	**true**	**true**

4 Integration Testing

A condition data flow diagram (CDFD) integrates condition processes by data flows and/or data stores. A sensible strategy for integration testing is to test every construct occurring in the CDFD in order to cover all the possible paths, where a path is a sequence of data flows from a *starting condition process* to a *terminating condition process*. A starting condition process is a condition process whose input data flows are not outputs of any other condition process in the same CDFD. It can also be a *source* (i.e., a condition process without any input data flow). A terminating condition process is a condition process whose output data flows are not inputs to any other condition process in the same CDFD. It can also be a *sink* (i.e., a condition process without any output data flow).

Our approach to testing a construct is to derive its proof obligation for ensuring the consistency and then test the proof obligation.

4.1 Testing Sequential Constructs

Figure 2(a) shows a sequential construct in a CDFD. We call this a sequential construct in the sense that only after firings of condition processes $A_1, A_2, \ldots, A_n$, is a firing of condition process B possible. If condition processes $A_1, A_2, \ldots, A_n$, and B are specified consistently, they must guarantee that when x_i satisfy the postcondition of A_i $(i = 1...n)$ under the constraint of its precondition, they will satisfy the precondition of B.

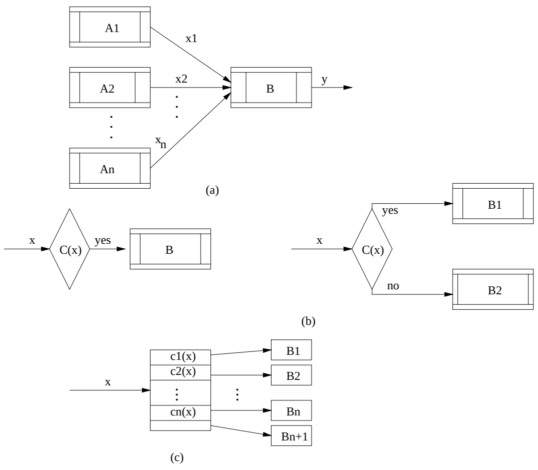

Fig. 2. The Constructs in CDFDs

Formally, this proof obligation is expressed as:

$$(pre_{A_1} \wedge post_{A_1}(x_1)) \wedge (pre_{A_2} \wedge post_{A_2}(x_2)) \wedge \cdots \wedge (pre_{A_n} \wedge post_{A_n}(x_n)) \Rightarrow pre_B \tag{2}$$

where each $post_{A_i}(x_i)$ is the sub-logical expression of the postcondition $post_{A_i}$ which contains variable x_i $(i = 1...n)$. For example, let $post_{A_1} \equiv x_1 > a$ **and** $x_1 < 10$ **or** $x_1 > a + 10$ **or** $y < a$, where a is an input to A_1 constrained by its precondition pre_{A_1}, then $post_{A_1}(x_1) \equiv x_1 > a$ **and** $x_1 < 10$ **or** $x_1 > a + 10$.

In comparison with testing of invariants and condition processes, testing this proof obligation is more effective in detecting faults. That is, we can *definitely* determine whether a test is a successful or failed test.

Definition 4.1 Let $T_d = \{T_1, T_2, \ldots, T_n\}$ be a test set for the proof obligation 2. If every single evaluation of this expression with each T_i $(i = 1...n)$ satisfies the condition that when $(pre_{A_1} \wedge post_{A_1}(x_1)) \wedge (pre_{A_2} \wedge post_{A_2}(x_2)) \wedge \cdots \wedge (pre_{A_n} \wedge post_{A_n}(x_n))$ evaluates to **true**, pre_B also evaluates to **true**, then the test with T_d is a failed test. Otherwise, if there exists any T_i such that pre_B evaluates to **false** whereas $(pre_{A_1} \wedge post_{A_1}(x_1)) \wedge (pre_{A_2} \wedge post_{A_2}(x_2)) \wedge \cdots \wedge (pre_{A_n} \wedge post_{A_n}(x_n))$ evaluates to **true**, the test with T_d is a successful test.

This definition provides a precise rule for determining whether or not a fault is detected by a test. For the sake of space, we do not give further illustration of this kind of testing by examples.

4.2 Testing Conditional Constructs

There are three kinds of conditional constructs in SOFL: IF-THEN, IF-THEN-ELSE and CASE, as given in Figure 2(b) and (c). As testing of these constructs share the same nature as for sequential constructs, we try to keep the discussion as brief as necessary.

IF-THEN The construct of this kind is illustrated by the graphical representation on the left hand side in Figure 2(b). Its proof obligation is:

$$pre \wedge post(x) \wedge C(x) \Rightarrow pre_B \tag{3}$$

where pre is the precondition of the preceding condition process; $post(x)$ is the sub-logical expression of its postcondition which contains variable x; and pre_B is the precondition of the condition process B.

The rule for determining a successful test or failed test for sequential constructs given in **Definition 4.1** can be applied to testing expression 3 if substituting $pre \wedge post(x) \wedge C(x)$ for $(pre_{A_1} \wedge post_{A_1}(x_1)) \wedge (pre_{A_2} \wedge post_{A_2}(x_2)) \wedge \cdots \wedge (pre_{A_n} \wedge post_{A_n}(x_n))$.

IF-THEN-ELSE This construct is illustrated by the graphical representation on the right hand side in Figure 2(b). Its proof obligation is

$$pre \wedge post(x) \wedge C(x) \Rightarrow pre_{B1} \tag{4}$$

$$pre \wedge post(x) \wedge \neg C(x) \Rightarrow pre_{B2} \tag{5}$$

Testing this proof obligation can be performed by testing expressions 4 and 5 respectively with the same method used for **IF-THEN** constructs.

CASE A CASE construct represents a multiple selection which is depicted by Figure 2(c). Its proof obligation is

$$pre \wedge post(x) \wedge C_i(x) \Rightarrow pre_{B_i} \tag{6}$$

$$pre \wedge post(x) \wedge \neg(C_1(x) \vee \cdots \vee C_n(x)) \Rightarrow pre_{B_{n+1}} \tag{7}$$

where $i = 1...n$.

If x satisfies condition $C_i(x)$, the precondition of the associated condition process B_i needs to be assured by the conjunction $pre \wedge post(x) \wedge C_i(x)$ so that the condition process B_i can be fired correctly. If x does not satisfy any of $C_1(x), \ldots, C_n(x)$, the precondition of condition process B_{n+1} must be assured for firing B_{n+1} correctly.

Again, testing this proof obligation can be performed by testing expressions 6 and 7 respectively with the same method used for **IF-THEN** constructs.

5 Testing Decompositions

A complete SOFL specification is a structured hierarchy of CDFDs, in which a condition process at one level may be decomposed into a CDFD at a lower level. The decomposition of a condition process defines how its inputs are transformed to its outputs in detail. While it needs to implement the specified functions of the high level condition process, the decomposition may also provide some additional functions under the constraint of the high level condition process specification in a strict *refinement* manner. That is, the decomposition must be a refinement of the high level condition process.

The rules for operational refinement have been well studied by researchers in the field [16, 3]. Those rules can also be applied to the decomposition of condition processes in SOFL.

Suppose a condition process OP is decomposed into a CDFD G. Let pre_{OP} and $post_{OP}$ denote the pre and postconditions of OP, respectively. Let pre_G and $post_G$ represent the pre and postconditions of G, respectively. The proof obligation for the decomposition

$$pre_{OP} \Rightarrow pre_G \tag{8}$$

$$pre_{OP} \wedge post_G \Rightarrow post_{OP} \tag{9}$$

must be satisfied by OP and G.

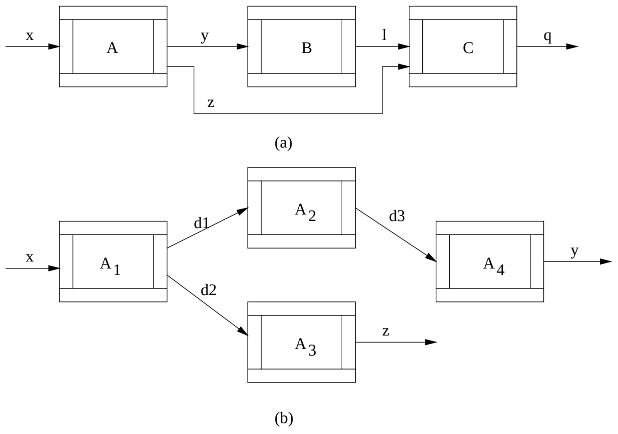

Fig. 3. An illustration of decomposition

A method for testing this proof obligation is that the same inputs generated for testing condition process A are used to test G. To keep the description concise, we only use a simple example to illustrate this method. Let condition process A in Figure 3(a) is decomposed into the CDFD, named G, in Figure 3(b). As G is a diagram, not a single condition process, when testing it we must first carry out unit testing for each condition process occurring in G, starting with the condition process A_1 and ending up with both A_3 and A_4, and then carry out integration testing to ensure the consistency between the condition processes. An essential idea we need to bear in mind in such a test is that *A must share the same inputs in the test cases with A_1 and share the same outputs with A_3 and A_4*. In analysis of the test results, certain conditions must be checked to decide whether a fault exists or not. Let T_d be a test set for A. Then these conditions are

$$pre_A \Rightarrow pre_{A_1}$$
$$pre_A \wedge post_{A_1} \Rightarrow pre_{A_2} \wedge pre_{A_3}$$
$$pre_{A_2} \wedge post_{A_2} \Rightarrow pre_{A_4}$$
$$(pre_{A_3} \wedge post_{A_3}) \wedge (pre_{A_4} \wedge post_{A_4}) \Rightarrow post_A$$

6 Conclusions and Future Research

This paper has proposed *specification testing* as a method for verifying and validating formal specifications, in order to help reduce the cost and risk of software development. The target is to deal with implicit or nonprocedural specifications and their integrations. The method is based on the combination of the ideas of formal proof and program testing. When testing a specification, the logical expressions which represent proof obligations for consistency are derived from the specification, and then the expressions are tested. Different criteria can be used to determine whether a test is a successful test, failed test, or non-confident test, depending on what the logical expressions represent. We have presented five strategies for testing logical expressions, which serve as guidelines for test case generation. We have also described how this testing method can be applied to SOFL to verify the consistency of specification components (e.g., invariants, condition processes), condition data flow diagrams (CDFDs), and decompositions of condition processes. Furthermore, validation of formal specifications by testing is also discussed.

Several important issues remain for our ongoing and future research. We are interested in investigating concrete and effective techniques for test case generation. To this end, we plan to conduct a relatively large scale case study of testing the specification for a "University Information System" which was developed using SOFL at Hiroshima City University. Another area of investigation is *automatic test case generation*. Furthermore, it is our belief that an effective tool support is crucial to the application of the technique for testing specification put forward in this paper. The author has been working with a research student on the construction of such a tool, and will continue to improve the tool in the near future.

Acknowledgements

I would like to thank Dr. Jian Chen of the University of South Australia for discussions on the initial idea of the research presented in this paper and Dr. Chris Ho-Stuart of Queensland University of Technology for improving the English presentation of the Introduction section. In particular, I would like to express my gratitude to Dr. John Fitzgerald of the University of Newcasle Upon Tyne and Dr. Jim Woodcock of Oxford University Computing Laboratory for reading through this paper in detail. Their constructive comments and suggestions have helped me improve significantly the quality of this paper. Special thanks are also due to the referees for their helpful comments.

References

[1] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[2] R. A. Kemmerer. Testing Formal Specifications to Detect Design Errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.

[3] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International(UK) Ltd., 1990.

[4] Antoni Diller. *Z: An Introduction to Formal Methods.* John Wiley & Sons, 1994.

[5] Lee J. White. *Software Testing and Verification*, volume 26. Academic Press, ADVANCES IN COMPUTERS, 1987.

[6] Roger. Duke, Gordon Rose, and Gordon Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[7] Jian Chen and Shaoying Liu. An Approach to Testing Object-Oriented Formal Specifications. In *Proceedings of the International Conference TOOLS Pacific 96: Technology of Object-Oriented Languages and Systems*, Melbourne, November 1996. TOOLs/ISE.

[8] I. J. Hayes. Specification Directed Module Testing. *IEEE Transactions on Software Engineering*, SE-12(1):124–133, January 1986.

[9] Marie Claude Gaudel Gilles Bernot and Bruno Marre. Software Testing based on Formal Specifications: a Theory and a Tool. *Software Engineering Journal*, pages 387–405, November 1991.

[10] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[11] Shaoying Liu, A. Jeff Offutt, Chris Ho-Stuart, Yong Sun, and Mitsuru Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, 24(1):337–344, January 1998. Special Issue on Formal Methods.

[12] Shaoying Liu, Masashi Asuka, Kiyotoshi Komaya, and Yasuaki Nakamura. An Approach to Specifying and Verifying Safety-Critical Systems with Practical Formal Method SOFL. In *Proceedings of Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'98)*, pages 100–114, Monterey, California, USA, August 10-14 1998. IEEE Computer Society Press.

[13] Shaoying Liu, Masaomi Shibata, and Ryuichi Sato. The Specification, Design and Implementation of a University Information System. Technical Report HCU-IS-99-005, Hiroshima City University, 1999.

[14] Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development.* Springer-Verlag London Limited, 1996.

[15] Chris Ho-Stuart and Shaoying Liu. A Formal Operational Semantics for SOFL. In *Proceedings of the 1997 Asia-Pacific Software Engineering Conference*, pages 52–61, Hong Kong, December 1997. IEEE Computer Society Press.

[16] Carroll Morgan. *Programming from Specifications.* Prentice-Hall International(UK) Ltd., 1990.

Appendix A

SOFL operators	Equivalent VDM-SL operators
forall[x **inset** D | P(x)]	$\forall x \in D \bullet P(x)$
exists[x **inset** D | P(x)]	$\exists x \in D \bullet P(x)$
notin	$\notin$
inset	$\in$
and	$\wedge$
or	$\vee$