International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:4, 2008

Generating State-Based Testing Models for Object-Oriented Framework Interface Classes

Jehad Al Dallal, and Paul Sorenson

***Abstract*—**An application framework provides a reusable design and implementation for a family of software systems. Application developers extend the framework to build their particular applications using hooks. Hooks are the places identified to show how to use and customize the framework. Hooks define the Framework Interface Classes (FICs) and the specifications of their methods. As part of the development life cycle, it is required to test the implementations of the FICs. Building a testing model to express the behavior of a class is an essential step for the generation of the class-based test cases. The testing model has to be consistent with the specifications provided for the hooks. State-based models consisting of states and transitions are testing models well suited to object-oriented software. Typically, hand-construction of a state-based model of a class behavior is expensive, error-prone, and may result in constructing an inconsistent model with the specifications of the class methods, which misleads verification results. In this paper, a technique is introduced to automatically synthesize a state-based testing model for FICs using the specifications provided for the hooks. A tool that supports the proposed technique is introduced.

***Keywords*—**Framework interface classes, hooks, state-based testing, testing model.

I. Introduction

An application framework provides a reusable design and implementation for a family of software systems [1]. Users of the framework complete or extend the framework to build their particular applications. Places at which users can add their own classes are called *hooks* [2]. To build an application using a framework, application developers create two types of classes: (1) classes that use the framework classes and (2) classes that do not. Classes that use the framework classes are called Framework Interface Classes (FICs) because they act as interfaces between the framework classes and the second type of the classes created by application developers. Building reusable test cases for the FICs and providing the test cases with the framework can potentially reduce the framework application testing time and increase application quality. Providing the frameworks with reusable test cases makes the frameworks more usable for and marketable to application developers.

As shown in Fig. 1, the input to the testing process of the FICs is the specifications of the FIC methods. Hook descriptions provide the specifications of the FIC methods in terms of pre- and post-conditions. There are two types of pre- and post-conditions: (1) construction ones and (2) execution ones and these are illustrated in the example in Fig. 2. The construction pre- and post-conditions are the constraints that must be satisfied before and after the hook is used, respectively, and they are identified in the hook description by keywords such as *Object*, *Class*, and *Operation*. The execution pre- and post-conditions are the dynamic constraints that must be satisfied before and after the methods defined in the hook are executed, respectively. The construction pre- and post-conditions, in contrast with the execution ones, do not describe the behavior of the methods defined in the hooks and, therefore, cannot be used in synthesizing the behavioral model of the FIC. The execution pre-conditions are described in terms of class instance variables and method input parameters. The execution post-conditions are described in terms of class instance variables, input parameters, output parameters, method return values, and method thrown exceptions. More precisely, the execution method specifications (i.e., pre- and post-conditions) check: (1) whether class instance variables, method input parameters, method output parameters, and method return values are within the allowed domain of values, and (2) whether the relationships among the values of the class instance variables, method input parameters, method output parameters, and method return values are satisfied.

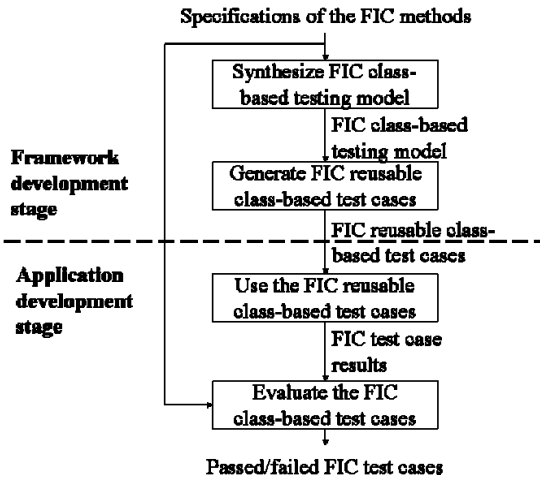

Fig. 1 The FIC testing process

To help understand the relationship between the hooks and the FICs let us examine a concrete example. Fig. 2 shows the description of the *Initialize Account* hook of a banking framework. In the *Changes* section of the hook, the FIC called *NewAccount* is introduced. The hook specifies also one of the

Jehad Al Dallal is with Department of Information Sciences, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait (e-mail: jehad@cfw.kuniv.edu).

Paul Sorenson is with Department of Computing Science, University of Alberta, Edmonton, AB. T6G 2H1, Canada (e-mail: sorenson@cs.ualberta.ca).

FIC methods, which is the constructor method *NewAccount(int amount)*. In the *Pre-conditions* and *Post-conditions* sections of the hook description, the pre-conditions and post-conditions of the constructor method are specified. The first stated post-condition is a construction post-condition because it describes a condition that must be satisfied when the hook is used and it is identified by the keyword *Operation*. The other pre- and post-conditions are execution ones because they describe the conditions that must be satisfied, respectively, before and after the constructor method is executed. To determine the behavior of a FIC we have to consider all the framework hooks that specify the FIC methods. For the rest of this paper, unless stated otherwise, all pre- and post-conditions referred to are of type execution, because these are the conditions in which we are most interested in building test cases.

Name: Initialize Account
Requirement: Initialize an account (i.e., set the currency and bank branches).
…
Pre-conditions: amount>=0;
Changes:
NewAccount.NewAccount(int amount) extends Account.Account(int amount);
…
Post-conditions:
1. Operation NewAccount. NewAccount (int);
2. NewAccount.balance>=0;
3. ! NewAccount.frozen;
4. NewAccount.getUpdate()< NewAccount.MaxPeriod

…

Fig. 2 The description of the *Initialize Account* hook of a banking framework

In our concrete example, the hooks of the banking framework define several public methods for the *NewAccount* FIC. The methods are *NewAccount*, *balance*, *deposit*, *withdraw*, *freeze*, *unfreeze*, *activate*. The pre- and post-conditions of the *NewAccount* FIC method defined in the hooks are listed in Table I. From the pre-conditions and post-conditions, we synthesize the states and transitions of the *NewAccount* class.

For example, in Table I, *balance()* is a method that has no pre-conditions which means that it can be called at any time during the object life cycle. *MaxPeriod* is a static instance variable, *amount* is a local variable, and the variables *balance*, *frozen*, the return of the *balance()* method, and the return of the *getUpdate()* method are non-static instance variables. The *balance()* method returns the value of the *balance* instance variable. The *getUpdate* method calculates the difference between the current date and the last activity date.

Despite the fact that the method specifications hold the specifications for the class behaviors, researchers seem to limit the method specification use to support the automated detection of software failures and the isolation of faults, and to generate method-based test cases. We are not aware of any work that uses the method specifications to generate class-based test cases. For example, in [3], to test a class behavior, class behavior testing models (e.g., state-transition model or UML statechart) used to generate the test drivers have to be pre-provided, and the method specifications are used only as testing oracles. Hand-construction of the class behavior testing model is expensive, error-prone, and may result in constructing an inconsistent model with the specifications of the class methods, which misleads verification results.

In this paper, a new technique is introduced to automatically synthesize the state-transition testing model of the FIC sequential class behavior from the specifications of the class methods. This reduces considerably the class testing cost and the chance of errors. The result is a state-based testing model that is consistent with respect to the specifications of the class methods. Therefore, using the introduced state-transition model synthesis technique, only the specifications of the FIC methods have to be provided to test the FIC behavior. The state-transition model is synthesized automatically from the method specifications. After that, a specification-based testing technique can be applied to derive the test drivers (i.e., implementations of the test cases) from the synthesized state-transition model. Finally, the test drivers are executed and the method specifications are used as testing oracles to evaluate the actual results of the test cases as pass or no pass. Fig. 3 compares the testing process that uses our proposed modeling technique and the one that does not (e.g., [4], [5], and [6]). In process (a), the FIC testing model as well as the FIC method specifications has to be provided to test the FICs, while in process (b), only the FIC method specifications have to be provided to test the FICs and the FIC testing model is synthesized internally in the process.

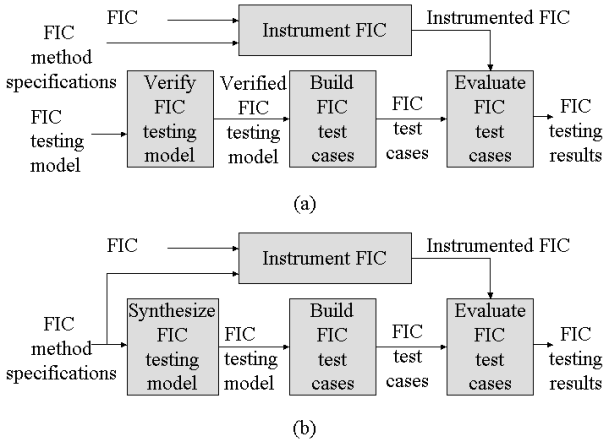

Fig. 3 FIC behavior testing process using method specifications: (a) without using the proposed modeling technique and (b) with using the proposed modeling technique

The paper is organized as follows. Section II, discusses the related work. In Section III and Section IV, the proposed techniques to extract the class behavior states and transitions, respectively, from the method specifications are described. Finally, Section V provides conclusions and a discussion of future work.

TABLE I
THE PRE-CONDITIONS AND POST-CONDITIONS OF THE *NEWACCOUNT* CLASS METHODS

Method	Pre-conditions	Post-conditions
NewAccount (amount)	amount>=0	NewAccount.balance>=0 && ! NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod
balance()		
deposit (amount)	! NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod	NewAccount.balance=amount+ NewAccount.balance && ! NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod
withdraw(amount)	NewAccount.balance>=0 && ! NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod	NewAccount.balance= NewAccount.balance-amount && ! NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod
freeze()	! NewAccount.frozen && NewAccount.balance>=0	NewAccount.frozen && NewAccount.balance>=0
unfreeze()	NewAccount.frozen && NewAccount.balance>=0	! NewAccount.frozen && NewAccount.balance>=0 && NewAccount.getUpdate()< NewAccount.MaxPeriod
activate()	NewAccount.balance>=0 && !NewAccount.frozen &&NewAccount.getUpdate()>= NewAccount.MaxPeriod	NewAccount.balance>=0 && !NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod

II. RELATED WORK

Researchers identified two benefits of using the method specifications (i.e., pre- and post-conditions) (1) evaluating the results of the test cases, and (2) building test cases to test the methods of classes. The method specifications and the class invariants are called contracts. In [7], contracts are used to evaluate the results of the test cases. It is shown that contracts detect large percentage of failures (roughly 80% of the faults detected using hard-coded oracles). Moreover, it is shown that the percentage of the detected faults depends on the precision of the contracts. Baudry et al, [8] showed that the quality of the contracts is more important than their quantity. In [7], it is found that the effort involved in isolating a fault improves eight folds between programs without contracts as compared to the ones with contracts.

There are several tools and languages introduced to support the use of the contracts. Design-by-Contract (DbC) [9] is used to specify the contracts in Eiffel language. Jcontract [10] and iContract [11] are tools used to implement DbC for Java programs. In both tools, contracts written in DbC are specified as Java comments. The iContract tool instruments the Java program with extra code to check the contracts. The instrumented Java program can then be compiled as usual with a Java compiler. The Jcontract tool is provided with a compiler that checks the DbC specifications and instruments the *.class* file with extra bytecodes to check the contracts at run time.

In [3], Java Modeling Language (JML) [12] and [13] is used to specify the contracts for Java methods. In this work, JML is integrated with Junit framework [14] to test the Java methods. JML is also used in the Korat framework [15], where the method specifications are used to automatically generate test cases for Java methods and to check the correctness of the outputs. JTest [16] is a tool that uses DbC contracts to automatically generate test cases for Java methods and to check the correctness of the outputs.

III. SYNTHESIZING THE STATES OF A FIC

To synthesize the states of a FIC, we have to construct the condition/instance-variable table. In the table, the columns and rows represent the non-static instance-variables of the FIC and the pre-condition/post-conditions of the FIC methods that contain conditions involving the non-static instance variables, respectively. In the table, we consider only the non-static instance variables. The values of the static instance variables do not change during the object life cycle and, therefore, they do not contribute in determining the object states. Table II shows the condition/instance-variable table extracted from Table I.

Finally, the table has to be optimized by eliminating redundant rows and unnecessary information. To optimize the table follow these steps

Step 1: Delete any clause that depends on a dynamic variable (i.e., a variable that its value is not assigned at compilation time or can change during the object life-cycle) because the combinations of the instance variable values that represent a state of a class are determined at compilation time and do not change during the object life-cycle. The deleted clauses are considered later in the transition synthesis process.

Step 2: Delete redundant rows.

Step 3: If the combinations of instance variable values in a row r_1 overlap with the combinations of instance variable values of another row r_2, replace r_1 and r_2 with three rows: the first one contains the combinations of instance variable values contained in r_1 and not contained in r_2, the second one contains the combinations of instance variable values contained in r_2 and not contained in r_1, and the third row contains the overlapping combinations of instance variable values.

Step 4: Iterate through steps 2 and 3 until no redundant rows and no rows for which Step 3 can be applied exist.

TABLE II
THE CONDITION/INSTANCE-VARIABLE TABLE EXTRACTED FROM TABLE I

Condition identifier	Source of the condition	Non-static instance-variables		
		frozen	**balance**	**getUpdate()**
1	post-condition of the *NewAccount* method, pre-condition of the *withdraw* method, post-condition of the *unfreeze* method, and post-condition of the *activate* method	false	>=0	<MaxPeriod
2	Pre-condition of the *deposit* method	false		<MaxPeriod
3	Post-condition of the *deposit* method	false	balance+amount	<MaxPeriod
4	Post-condition of the *withdraw* method	false	balance-amount	<MaxPeriod
5	Pre-condition of the *freeze* method	false	>=0	
6	Post-condition of the *freeze* method and pre-condition of the *freeze* method	true	>=0	
7	Pre-condition of the *activate* method	false	>=0	>=MaxPeriod

Each row in the optimized table represents a state of the object of that class during its life-cycle based on the instance variable value combination shown in the row. Step 1 ensures that the states do not change during the object life-cycle (i.e., a basic property of the states of a class state-based model as described in [4]) and, therefore, each remaining clause in the table has boundary values determined at compilation time. The second step ensures that there are no redundant states in the synthesized model. Step 3 ensures that each synthesized state is exclusive (i.e., there is no overlapping states). Determining the overlapping combinations of instance variable values contained in two rows is straightforward because the condition clauses in the rows have fixed boundary values as ensured in Step 1. Finally, Step 4 holds the stopping criterion for the state synthesis process.

When the optimization rules are applied on Table II, according to Step 1, 'balance+amount' and 'balance-amount' are deleted from rows 3 and 4, respectively, because they depend on dynamic variables. This makes rows 3 and 4 redundant with row 2 and, therefore, rows 3 and 4 are deleted according to Step 2. The combinations of instance variable values contained in row 1 overlap with the combinations of instance variable values contained in row 2. Therefore, to avoid the creation of overlapping states, rows 1 and 2 are substituted with three rows. The first row contains the combinations of instance variable values contained in row 1 and not contained in row 2. This row is ignored because it does not contain any combination of instance variable values (i.e., all the combinations of instance variable values contained in row 1 overlap with the combinations of instance variable values contained in row 2). The second row contains the combinations of instance variable values contained in row 2 and not contained in row 1 (i.e., the combinations of instance variable values that has *balance*<0). The third row contains the overlapping combinations of instance variable values of rows 1 and 2 (i.e., the combinations of instance variable values that has balance *>=0*). Finally, the combinations of instance variable values contained in the third row formed in the previous step overlap with the combinations of instance variable values contained in row 5. Therefore, the two rows are replaced with three rows. The first row is ignored because it does not contain any combination of instance variable values (i.e., all of the combinations of instance variable values contained in the third row formed in the previous step overlap with the combinations of instance variable values contained in row 5). The second row has *getUpdate() >=MaxPeriod* and the third row has *getUpdate() <MaxPeriod*. This results in having the optimized table shown in Table III. In this table, each row represents a state that corresponds to the instance variable value combinations given in the row. The combinations of instance variable values in each row are called the *state-invariants* [4].

TABLE III
OPTIMIZED TABLE CONSTRUCTED BY APPLYING OPTIMIZATION RULES ON TABLE II

State identifier	Instance-variables		
	frozen	**balance**	**getUpdate()**
1	false	>=0	<MaxPeriod
2	false	<0	<MaxPeriod
3	false	>=0	>=MaxPeriod
4	true	>=0	

IV. SYNTHESIZING THE TRANSITIONS OF A FIC

To extract the transitions that model the legal behavior of a FIC, we have to map the pre-conditions and post-conditions of the FIC methods to the extracted state-invariants. Each state in which its state-invariants satisfy the pre-conditions of a method is a source state for the transition associated with the method call. Moreover, each state in which its state-invariants satisfy the post-conditions of a method is a destination state for the transition associated with the method call. The procedure shown in Fig. 4 explains the mapping process and shows how to extract the predicates and actions of the transitions. The source state of the constructor method is by default the alpha state. If no destructor method is specified in the class, an unlabeled transition has to be added from each state, other than the alpha state, to the omega state.

Inputs: Invariants of the FIC states and the pre-conditions and post-conditions of the FIC methods.
Output: Transitions of the FIC state-based model.
Procedure:
1. *for* each FIC method *do*
2. Search for all states whose state-invariants satisfy the pre-conditions of the method.
3. Search for all states whose state-invariants satisfy the post-conditions of the method.
4. Create a transition from each state found in Step 2 to each state found in Step 3 and associate the method name with the transition as an event.
5. *for* each transition created in *Step 4 do*
6. *if* the set of pre-conditions of the method is a superset of the set of state-invariants of the source state of the transition *then* add the non-overlapped portion of the set as predicates to the transition.
7. *if* the set of post-conditions of the method is a superset of the set of state-invariants of the destination state of the transition *then* add the non-overlapped portion of the set as actions to the transition.
8. *if* there is another transition that has the same source state, event, and predicates *then* add to each of the transitions the difference between the post-conditions of the method called in the transition and the state-invariants of the destination states as predicates to the transitions.

Fig. 4 Construction process of the transitions of the FIC specification model

When the procedure shown in Fig. 4 is applied on tables I and III, the transitions shown in Table IV are extracted. For example, since the balance method has no pre-conditions and its post-condition satisfies all the invariants of all states, a self-loop transition associated with balance() event is added to each state (other than alpha and omega). The post-condition of the balance method is not specified in any state and, therefore, it is added as an action to all the self-loop transitions. For the withdraw method, the pre-conditions satisfy the invariants of state 1 and the post-conditions satisfy the invariants of states 1 and 2. Therefore, two transitions associated with the withdraw event are added as shown in Table IV. Note that the set of pre-conditions of the method is the same as the invariants of state 1, which causes no predicates to be added at this step. The set of post-conditions of the withdraw method includes "balance=balance-amount" which is not included in the state invariants of states 1 and 2. Therefore, the post-condition is added to both transitions as actions. Finally, since state 1 now has two outgoing transitions that have the same labels, the difference between the invariants of the destination states of the transitions has to be added as predicates to the transitions. The difference between states 1 and 2 is that state 1 has balance>=0, while state 2 has balance<0. Therefore, "balance (at destination state) >=0" has to be added to one of the transitions (from state 1 to state 1) and "balance (at destination state) <0" has to be added to the other transition (from state 1 to state 2). Since the predicates are checked at the source states, we can substitute "balance (at destination state)" by "balance-amount".

TABLE IV
TRANSITIONS OF THE *NEWACCOUNT* FIC EXTRACTED USING THE PROCEDURE SHOWN IN FIG. 4

Transition identifier	Source state identifier	Destination state identifier	Transition event	Transition predicates	Transition actions
1	alpha	1	NewAccount	amount>=0	
2	1	1	balance		return balance
3	2	2	balance		return balance
4	3	3	balance		return balance
5	4	4	balance		return balance
6	1	1	deposit	balance+amount>=0	balance=balance+amount
7	1	2	deposit	balance+amount<0	balance=balance+amount
8	2	1	deposit	balance+amount>=0	balance=balance+amount
9	2	2	deposit	balance+amount<0	balance=balance+amount
10	1	1	withdraw	balance-amount>=0	balance=balance-amount
11	1	2	withdraw	balance-amount<0	balance=balance-amount
12	1	4	freeze		
13	3	4	freeze		
14	4	1	unfreeze	balance>=0	
15	3	1	activate		
16	1	Omega			
17	2	Omega			
18	3	Omega			
19	4	Omega			

Since the transition synthesis method uses the method specifications to synthesize the transitions, it is limited to event-driven transitions (i.e., transitions that have associated events). For non-event-driven transitions, it is required to determine the source and destination states first. The state-invariants of the destination state, which are different than the state-invariants of the source state, are then added as predicates to the transition. For the NewAccount class example, we have identified two non-event-driven transition examples as shown in Table V. The invariant of state 3, which is different than the invariant of state 1, is "getUpdate()>=MaxPeriod". Therefore, this difference is added as a predicate to the non-event-driven transition that has the states 1 and 3 as source and destination states, respectively. The same situation applies for the transition that has the states 4 and 3 as source and destination states, respectively.

TABLE V
NON-EVENT-DRIVEN TRANSITIONS OF THE *NEWACCOUNT* FIC

Transition identifier	Source state identifier	Destination state identifier	Transition predicates	Transition actions
20	1	3	getUpdate()>=MaxPeriod	-
21	4	3	getUpdate()>=MaxPeriod && !frozen	-

Fig. 5 shows the synthesized state-transition model represented in a State Transition Diagram (STD). In the diagram, states and transitions are represented by nodes and edges, respectively. To make the diagram more understandable, meaningful names can be associated to the states. In Fig. 5, states numbered 1, 2, 3, and 4 are named *Open*, *Overdrawn*, *Inactive*, and *Frozen*, respectively. The transitions are labeled by their identifiers shown in Tables IV and V.

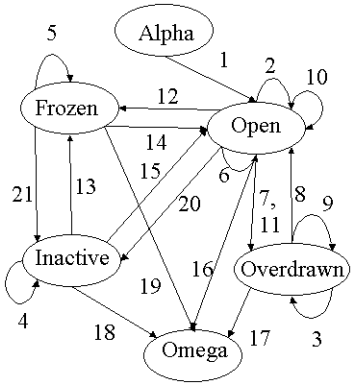

Fig. 5 STD of the synthesized state-transition model of the *NewAccount* FIC

V. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a technique to synthesize the state-transition model for the sequential class behavior using the specifications of the FIC methods provided in the hook descriptions. The technique first synthesizes automatically the states of the model. Then it synthesizes automatically the event-driven transitions of the model and their attributes (i.e., event, predicates, and actions). Given the source and destination states of the non-event-driven transitions, the technique finds automatically the attributes of the transitions. As a result, the test drivers used to test the sequential behavior of the FIC can be generated from the contracts almost for free.

Moreover, using the proposed synthesis technique the time required to verify the correctness of the model is eliminated. The proposed technique does not guarantee synthesizing a free of infeasible paths model. Infeasible paths are the ones that cannot be executed. To solve this problem, we have to either detect the infeasible paths and avoid using them in generating the test drivers [17], or we have to ignore any test driver that has violated pre-conditions [3]. The proposed technique focuses on modeling classes that have sequential behaviors. Further research is required to model classes that have concurrent behaviors. To model such classes, synchronization contracts [18] can be used. A prototype tool is developed to automate the FIC testing process shown in Fig. 1. However, the first step of the process is not implemented yet. Further work is required to extend the tool to fully automate the testing process.

REFERENCES

[1] K. Beck and R, Johnson. Patterns generated architectures, *Proc. of ECOOP 94*, 1994, 139-149.
[2] G. Froehlich. Hooks: an aid to the reuse of object-oriented frameworks, *Ph.D. Thesis, University of Alberta, Department of Computing Science*, 2002.
[3] Y. Cheon and G. Leavens, A simple and practical approach to unit testing: the JML and JUnit way, Proc. of the 16th European Conference on Object-Oriented Programming (ECOOP2002), June 2002, pp. 231-254.
[4] R. Binder. Testing object-oriented systems, Addison Wesley, 1999.
[5] J. Offut and A. Abdurazik, Generating tests from UML specifications, Second International Conference on the Unified Modeling Language (UML99), Fort Collins, CO, October 1999, 416-429.
[6] A. Abdurazik, P. Ammann, W. Ding, and J. Offutt, Evaluation of three specification-based testing criteria, Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00), Tokyo, Japan, September 2000, 179-187.
[7] L. Briand, Y. Labiche, and H. Sun, Investigating the use of analysis contracts to support fault isolation in object-oriented code, International Symposium on Software Testing and Analysis ISSTA, Rome, Italy, July 2002.
[8] B. Baudry, Y. LeTraon, and J.-M. Jézéquel, Robustness and diagnosability of OO-systems designed by contracts, *Proceedings of Metrics'01*, London, UK, April 2001.
[9] B. Meyer, Design by contracts, IEEE Computer, 1992, Vol. 25(10), 40-52.

[10] Jcontract, http://www.parasoft.com/jsp/products/home.jsp?product=Jcontract, ParaSoft Corporation, July 2006.
[11] iContract: the Java Design-by-Contract tool, http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html, July 2006.
[12] G. Leavens, A. Baker, and C. Ruby, Preliminary design of JML: a behavioral interface specification language for Java, TR 98-06p, Iowa State University, Department of Computer Science, August 2001.
[13] G. Leavens, A. Baker, and C. Ruby, JML: a notation for detailed design. In H. Kilov, B. Rupe, and I. Simmonds, editors, behavioral specifications of Businesses and Systems, chapter 12, Kluwer, 1999, pp. 175-188.
[14] Junit, http://junit.sourceforge.net/, July 2006.
[15] C. Boyapati, S. Khurshid, and D. Marinov, Korat: Automated Testing Based on Java Predicates, International Symposium on Software Testing and Analysis ISSTA, Rome, Italy, July 2002.
[16] Jtest, http://www.parasoft.com/jsp/products/home.jsp?product=Jtest, ParaSoft Corporation, July 2006.
[17] J. Offutt and J. Pan, Automatically detecting equivalent mutants and infeasible paths, The Journal Of Software Testing, Verification, and Reliability, 7(3), September 1997, pp 165-192.
[18] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, Making components contract aware, *IEEE Computer*, 13(7), July 1999.