An Approach to Checking Consistency between UML Class Model and Its Java Implementation

Hector M. Chavez, Wuwei Shen, *Member, IEEE*, Robert B. France, Benjamin A. Mechling, and Guangyuan Li

Abstract—Model Driven Engineering (MDE) aims to expedite the software development process by providing support for transforming models to running systems. Many modeling tools provide forward engineering features, which automatically translate a model into a skeletal program that developers must complete. Inconsistencies between a design model and its implementation, however, can arise, particularly when a final implementation is developed dependently on the code from which it was generated. Manually checking that an implementation conforms to its model is a daunting task. Thus, an MDE tool that developers can use to check that implementations conform to their models can significantly improve a developer's productivity. This paper presents a model-based approach for testing whether or not an implementation satisfies the constraints imposed by its design model. Our model-based testing approach aims to efficiently reduce the test input space while supporting branch coverage criteria. To evaluate the approach's ability to uncover inconsistencies, we developed a prototypical tool and applied it to the Eclipse UML2 projects. We were able to uncover inconsistencies between the models and their implementations using the tool.

Index Terms—Class diagrams, UML, Java, model checking

1 Introduction

Computer use is pervasive in our daily life and the increasing demand for complex computer applications that play critical roles in society makes software development much more complicated. Model Driven Engineering (MDE) has been proposed to tackle the problems facing the software engineering community. In particular, MDE research has led to the development of tools that can automatically translate a model into skeletal code, leaving the method body implementation to developers. Especially when timelines are tight, manual implementation may easily violate model constraints due to human error or a misunderstanding of the model. In these cases, checking consistency between a design model and its implementation is needed to support the role of models as contracts for implementations as they evolve. Consistency checking can also assist in the understanding of an implementation and its design, and, consequently, help enable effective use of model driven design approaches.

In some domains, such as embedded software systems, mature consistency checking technologies exist primarily because of the close semantic gap between the modeling and the implementation languages typically used in these domains. For example, Reactis can automatically check whether a C program conforms to a Simulink design model [1]. In the more general object-oriented (OO) software development area, however, consistency checking between an OO design model and its implementation can be challenging because of the wider semantic gaps between OO modeling and programming languages. Specifically, the implementation of an OO modeling concept can vary between programming languages. For example, an association in UML is a high-level abstraction that does not have a direct counterpart in OO programming languages. The implementation of an association can vary from an array, to a list, or even a map depending on a programmer's preference.

In this paper we describe an approach to checking the consistency between a model expressed in a general-purpose modeling language, the Unified Modeling Language (UML) [2], and a Java implementation. We observe that one of the reasons of why MDE has gained the popularity in the software community is because many forward engineering tools (Fig. 1a) support the translation of UML class diagram (Fig. 1i) into Java (Fig. 1ii). Specifically, these tools generate not only the skeletal structure of Java classes according to its UML counterparts but also the implementation of some model features as a part of the output shown in Fig. 1ii. For example, object creation can be implemented with basic constructors. Association link creation and modification can be implemented with standard getter and setter methods for class fields. For other behaviors, code generators produce method signatures with empty bodies that developers must implement to obtain a final implementation shown in Fig. 1iii. Generating skeletal code relieves developers of manually producing code for mundane behaviors and in turn circumvents human-generated errors for these behaviors (assuming these behaviors are not changed once generated). Thus, forward engineering tools

- *H.M. Chavez, W. Shen, and B.A. Mechling are with Computer Science, Western Michigan University, Kalamazoo, MI 49008. E-mail: h6chavezchav@wmich.edu., {wuwei.shen, benjamin.a.mechling}@wmich.edu.*
- *R.B. France was with Computer Science, Colorado State University, Fort Collins 80523, United States. E-mail: france@cs.colostate.edu.*
- *G. Li is with the State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. E-mail: ligy@ios.ac.cn.*

Manuscript received 10 Mar. 2014; revised 25 Sept. 2015; accepted 30 Sept. 2015. Date of publication 7 Oct. 2015; date of current version 22 Apr. 2016. Recommended for acceptance by L. Baresi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2488645

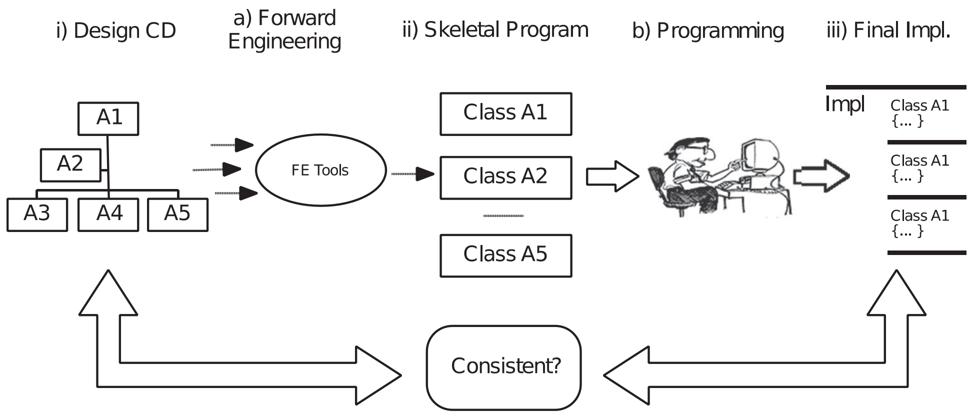


Fig. 1. The consistency problem in the context of MDE.

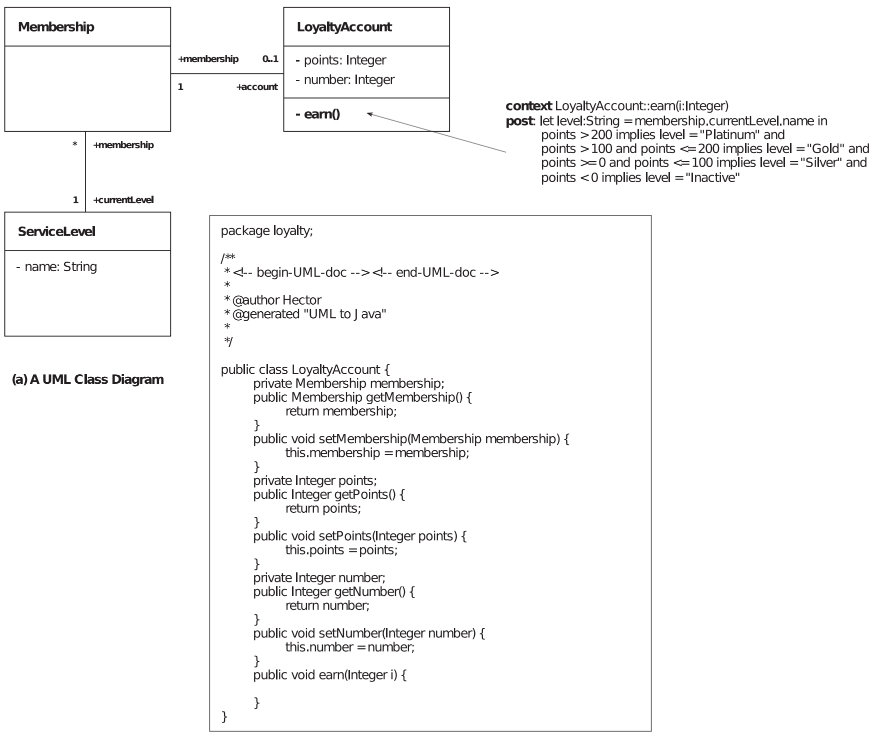
are favored by many third-party developer teams. Code generation intends to allow developers to focus on the implementation of void method bodies to produce a final implementation (Fig. 1b) more rapidly and reliably. Ironically, if the development team does not understand the forward engineering tool well, the number of errors can actually increase. For instance, a programmer's misunderstanding of the generated code's behavior or purpose can lead to the introduction of inconsistencies, that is, the programmer implements a software system based on a faulty interpretation of the generated framework (code). The increasing number of errors in a final implementation which is developed based on the generated framework by forward engineering tools leads to consideration of consistency checking tools that determine whether a final implementation of Fig. 1iii and design model of Fig. 1i are properly aligned.

In this paper, a Java implementation refers to the final implementation built on the Java skeletal code generated by some forward engineering tools. We assume that a generated Java skeletal code correctly implements the basic structures of a class diagram such as inheritance and association. Furthermore, it is assumed that a final implementation does not alter the behaviors of the generated skeletal code such as object creation and association link creation and modification. An inconsistency, therefore, is said to be present when constraints specified in a class diagram are violated by the behaviors of the void methods provided by programmers. In this case, we say the Java implementation does not conform to the class diagram (as a design model). Generally, constraints in a class diagram can come from two sources. The first source is the UML specification, which specifies the UML syntax and static semantics. Many such constraints in the UML specification are expressed using the Object Constraint Language (OCL) [3]. For example, the following well-formedness constraint associated with the metaclass *Property*—"*A multiplicity of a composite aggregation must not have an upper bound greater than 1*" [2] (p. 126)—is expressed as an OCL expression in the UML specification. When a Java field is implemented as a composite property in a class diagram, one must check that the above constraint in [2] holds. The second source is the set of application-specific constraints specified by developers during the development process. For example, Larsson et al. used OCL to express invariants and operation *pre-conditions* and *post-conditions* for the Java Card API [4]. Such constraints expressed in a class model enhance the quality of a software system since they can help catch more software faults during software testing.

An example of a class diagram with OCL constraints and generated skeletal code is shown in Fig. 2. Fig. 2a shows a class diagram with an OCL constraint attached to operation *earn()* in the class *LoyaltyAccount* (the diagram is excerpted from the *Royal and Loyal* example [3]). Fig. 2b shows the Java skeletal program generated by Rational Software Architect (RSA) [5], an MDE tool. As with many forward engineering tools, RSA leaves the responsibility of defining method bodies to programmers, while each property in a class diagram is translated to a Java field with associated setter and getter methods.

While the skeletal program in Fig. 2b is quite simple, programmers often encounter more complicated skeletal code structures by some forward engineering tools. For example, the Eclipse Modeling Framework (EMF) [6] generates skeletal programs using the Ecore metamodel and includes information specific to the metamodel. While some programmers consider this information to be extraneous, it actually has some advantages for design purposes such as making the generated program easier to be extended and adapted. For instance, the adapter factory class provides a convenient way to build factories that demand specific adapters for modeled objects [7]. The extra information, however, can make it difficult for programmers to understand the generated code and, thus, make the further development based on the skeletal code tedious and error-prone. In these cases, an approach that checks consistency between a class diagram and its final Java implementation can help a programmer determine whether their changes in the implementation are still consistent with respect to the model from which the skeleton was originally generated.

In this paper, we address the UML class model/Java program consistency problem outlined above. We propose a novel automated testing-based approach to support Conformance Checking between UML and Java, called CCUJ for short. CCUJ uses the concolic execution to efficiently prune the test input space [6]. More specifically, our approach attempts to check whether the state that is produced after calling a Java program on a test case is consistent with all the constraints associated with the UML design class diagram. The contributions of our approach are given below:



```
context LoyaltyAccount::earn(i:Integer)
post: let level:String = membership.currentLevel.name in
        points > 200 implies level = "Platinum" and
        points > 100 and points <= 200 implies level = "Gold" and
        points >= 0 and points <= 100 implies level = "Silver" and
        points < 0 implies level = "Inactive"
```

(a) A UML Class Diagram

```
package loyalty;

/**
 * <!-- begin-UML-doc --> <!-- end-UML-doc -->
 *
 * @author Hector
 * @generated "UML to J ava"
 *
 */

public class LoyaltyAccount {
        private Membership membership;
        public Membership getMembership() {
                return membership;
        }
        public void setMembership(Membership membership) {
                this.membership = membership;
        }
        private Integer points;
        public Integer getPoints() {
                return points;
        }
        public void setPoints(Integer points) {
                this.points = points;
        }
        private Integer number;
        public Integer getNumber() {
                return number;
        }
        public void setNumber(Integer number) {
                this.number = number;
        }
        public void earn(Integer i) {

        }
}
```

(b) A program generated by RSA for CD in (a)

Fig. 2. An example of a class diagram and its Java implementation generated by RSA.

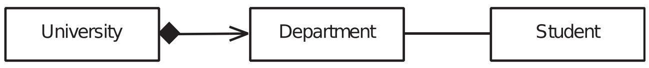
- We formalized the concolic execution via a simple Java-like language and then presented the main algorithm in CCUJ. A rigorous analysis about the main algorithm has been discussed;
- We expanded universal symbolic execution to the bytecode level to support collection of symbolic path conditions across methods, inter-method relationships;
- We employed an efficient pruning technique to ensure every test case value explores a different execution path in the execution of a method under test and its *post-method* generated from the OCL constraints in a class diagram;
- We discussed the impact of black box methods on the pruning technique utilized by symbolic execution since black box methods cannot be avoided in a large software system;
- We applied CCUJ to industry-strength projects (Model Development Tool UML2 v1.1.1 and UML2 v2.2.1). Several software faults in the UML2 projects were found and confirmed by the project developers. The faults were corrected in later versions.

This paper is an extended version of our previous conference paper [7]. The main difference between this paper and our previous conference paper are as follows. First, we formalize the consistency checking between a class model and its Java implementation. The formalization extends our previous work of testing a Java implementation against the constraints in a class model. Second, we further abstract our previous algorithm into one using the formalization and prove the correctness of the algorithm. Third, we discuss how a black box method affects the correctness of the symbolic testing. Fourth, we compare CCUJ with other approaches in terms of time performance and discuss the effectiveness of CCUJ via some mutation testing. Last, more test cases are conducted and a new software fault has been revealed by CCUJ. The new empirical results based on time performance strengthen the conclusion about CCUJ in our previous paper, i.e., CCUJ is not only effective but also efficient when checking consistency between a class model and its Java implementation. The remainder of the paper is organized as follows. Section 2 presents background on CCUJ. Section 3 demonstrates how CCUJ works with an illustrative example. In Section 4, we formalize the consistency checking and present the algorithm of CCUJ followed by the rigorous analysis. Section 5 presents several implementation issues behind CCUJ. We present case studies in Section 6 and discuss some related work in Section 7. Finally, we draw a conclusion and give future work in Section 8.

2 Background

Consistency checking between a UML class diagram and its Java implementation can be performed using either formal verification or *testing-based* validation techniques. Formal verification techniques have made some progress in past decades and are powerful, but they often do not scale well to large software models. CCUJ is a *testing-based* approach that leverages a program's model to efficiently generate test cases.

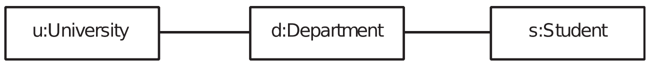
Before we dive into details about CCUJ, we define the terms used in this paper. A static defect in software is called a *software fault* [8]. The software whose implementation needs to be tested is called the *software under test*. The *input values* or *pre-state* necessary to complete some execution of the software under test are called *test case values*. A *test oracle* specifies the *expected results* or *post-state* for a complete execution and evaluation of the software under test. A test case



i) A Design Class Diagram Used to Generate Code

```
class University extends Root {
    …
    ArrayList<Department> depts.; // containees inherited from class Root
    public void set_Department(Department d) { depts.add(d);set_Containees(d);}
    …
    public void destroy() {
        ArrayList<Student> students = … // stores a list of students
        for (Student s:students) {
            if(get_Containees()<>null) {
                … // remove all references from s to elements in container
            } else
                return;
        } }
}
```

ii) A destroy() method to be tested



iii) An object diagram shown as a test case

Fig. 3. An example showing the problem caused by flipping a condition.

consists of test case values and *test oracles* for the software under test.

We have found that using traditional testing techniques to check the consistency between a class diagram and its Java implementation can be problematic. In general, traditional program testing suffers from two major shortcomings with respect to consistency checking. First, most testing techniques cannot guarantee the correctness of a program under test. More specifically, they optimistically hope that the execution of a faulty statement can expose a software fault in the absence of an explicit specification of correct behavior. Thus, most testing techniques adopt different coverage criteria to impose test requirements on test cases such as statement coverage that covers all statements including the faulty statements. Unfortunately, many errors in a program cannot be revealed in that testing techniques cannot guarantee the absence of errors. If a program does not have an assertion statement, then it is possible not to reveal an error even when a faulty statement is reached. A good example is given in the book by Ammann and Offutt [8]. In fact, consistency checking differs from the traditional testing since the former considers a constraint from a design model as a test oracle while the latter does not require such a constraint.

Second, many symbolic testing techniques that attempt code coverage by simply creating tests cases to cover all branches can have problems with partially generated code. Problems arise when advanced forward engineering tools implement a class diagram using auxiliary information or data structures to enforce the model's features that are not directly available in the language. To alter the path of execution some approaches may forcibly altering auxiliary information, causing the code to deviate from the original class diagram and, thus, possibly cause unexpected behavior such as a runtime error or a false positive against a test oracle. For example, consider the class diagram shown in Fig. 3i. Assume that a forward engineering tool generates a field *containees* in class *Root* inherited by all generated Java classes. Field *containees* of object *a* is the set of all objects that are owned by *a* in a composition relationship. The generated *containees* field is used to store object *a* with its all owned objects in one resource file so that object persistence is supported. Thus, in Fig. 3i, since class *University* is the owner end (black diamond) of a composition relationship with class *Department,* method *set_Department(Department d)* sets *d* to not only *depts* but also *containees.* However, field *containees* in the other two classes is never set since class *Student* is not a part of the composition and class *Department* is not an owner in the composition. Now, assume that method *destroy()*, shown in Fig. 3ii, is called on a test case whose structure is represented by the object diagram shown in Fig. 3iii. Then conditional statement *if(get_Containees()<> null)* cannot be flipped. If it is forcibly flipped to *false,* then the code under the *true* branch is not executed and a false positive is produced in this case.

2.1 The CCUJ Approach

CCUJ is a model-based testing approach that addresses the shortcomings described above. It uses branch coverage criteria and aims to efficiently prune the test input space by means of universal symbolic execution [6] integrated into concrete execution. To address the shortcomings of traditional program testing techniques, CCUJ makes use of the class diagram and identifies the variables that it should track in an implementation. Specifically, CCUJ targets the program variables used by programmers after code is generated from a design class diagram. When checking OCL constraints in a class diagram, CCUJ only considers a *pre-condition* and *post-condition* of an operation since a class invariant can be converted to conditions incorporated into each operation's *pre-condition* and *post-condition* in a class. Furthermore, because CCUJ aims to check a Java implementation against a design class diagram, the satisfaction of a *post-condition* as a test oracle is the main goal in CCUJ. Also, we note that a *pre-condition* is only used to produce correct test cases. Thus, we assume that a *pre-condition* can be converted to a formula sent to a Boolean Satisfiability solver (abbreviated SAT) to find a test case. While some research work has also used such a conversion [9], we outline how to extract a formula from a *pre-condition* using an algorithm similar to the one used by CCUJ. Thus, the generation of test cases via a SAT solver from *pre-conditions* written in OCL is not the focus of this paper. Instead, CCUJ only considers the multiplicity and navigability constrains of an association in a class diagram as operations *pre-condition*.

CCUJ employs a process that consists of a sequence of actions, which are illustrated in Fig. 4. Initially, CCUJ parses a class diagram to collect the information and stores the information in an internal data structure that can be easily accessed in later steps. Next, CCUJ retrieves *post-conditions* of all the methods in the diagram and translates them into Java post-methods inserted into the corresponding classes. After the generation of Java *post-methods*, CCUJ instruments all the code at the bytecode level to support dynamic testing. All these actions are done once and belong to the preliminary testing preparation step. Entering the dynamic testing step, CCUJ finds the first untested method. If an untested method *m* exists, CCUJ generates a test case and calls the method *m* followed by its *post-method* on the test case. If the *post-method* returns *false* CCUJ is done with the testing process since a fault is found. Otherwise, CCUJ investigates the symbolic path conditions of both method *m*

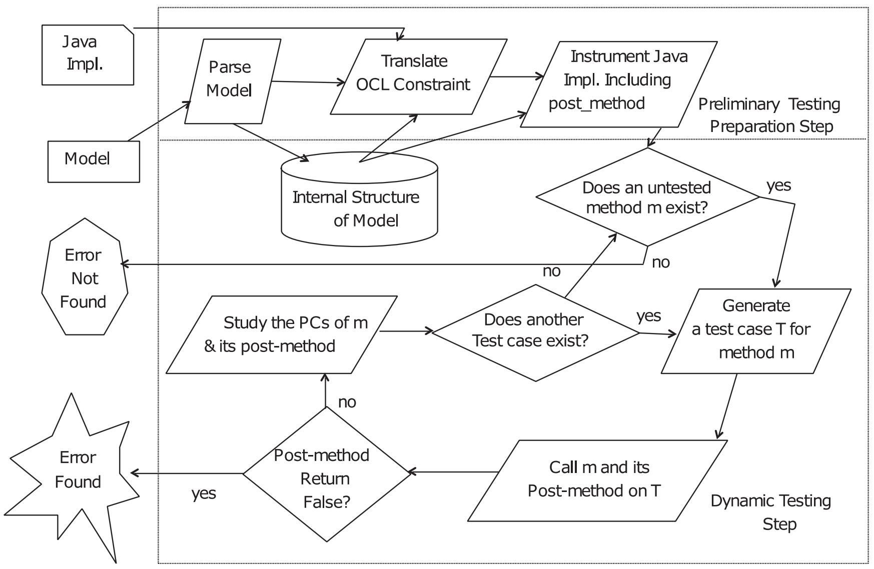

Java Impl.
Model
Parse Model
Translate OCL Constraint
Instrument Java Impl. Including post_method
Preliminary Testing Preparation Step
Internal Structure of Model
Does an untested method m exist?
yes
no
Error Not Found
no
Study the PCs of m & its post-method
Does another Test case exist?
yes
Generate a test case T for method m
no
Error Found
yes
Post-method Return False?
Call m and its Post-method on T
Dynamic Testing Step

Fig. 4. The CCUJ approach.

and its *post-method*. A SAT solver to determine whether a new test case exists to explore a different execution path on method *m* or its *post-method*. If such a test case exists, CCUJ generates the test case and calls *m* and its *post-method* again. Otherwise, CCUJ finds another untested method. If there is no untested method, CCUJ reports that no fault is found in the implementation.

The consistency checking between a class diagram and its final Java implementation supported by CCUJ can be reduced to checking whether a Java implementation conforms to a class diagram. Thus, the consistency checking can be more precisely defined as follows. *Given a design model consisting of a class diagram with OCL constraints, and a Java implementation, automatically generate a set of high-quality test cases to explore execution paths of the implementation to reveal behavior that does not conform to (the constraints specified in) the design model.* To this end, CCUJ investigates the relationship between the path constraints of the method under test and its post-method after each execution to ensure that a new test case value will explore a different execution path.

3 An Illustrative Example

We use the *Royal and Loyal* system example again to illustrate how CCUJ works. The OCL constraint shown in Fig. 2a is the *post-condition* to method *earn()*. The code generated by Rational Software Architect is partially shown in Fig. 2b, where each property is mapped to private class fields with setter and getter methods. We show how CCUJ can be used to check whether the implementation of *earn()* shown in Fig. 5a conforms to the class diagram shown in Fig. 2a. Specifically, we check if the implementation satisfies the only OCL constraint in the class diagram. In short, CCUJ takes as input a class diagram that includes method *earn()* and its OCL *post-condition*, shown in Fig. 2a, and its implementation, as shown in Fig. 5a.

In the following discussion, let Φ denote the class diagram-to-code translation schema employed by RSA. As a first step, CCUJ parses the class diagram to extract the corresponding OCL *post-condition* for the method under test, and it automatically generates the Boolean Java *post-method* *post_earn()* shown in Fig. 5b. Next, CCUJ uses the class diagram and the RSA translation schema Φ to match elements of the diagram with the implementation to produce a *minimal test case* for method *earn(i:Integer)*, shown in Fig. 6a. A *minimal test case* is a test case value that includes the smallest number of objects required to satisfy the model's constraints on the method's *pre-state*. To this end, CCUJ automatically produces the program shown in Fig. 6b, which calls the constructors and setter methods to generate the necessary objects and their links. The execution of the program produces the first *test case value* or *pre-state*. As the terms *test case value* and *pre-state* both represent the same concept they are

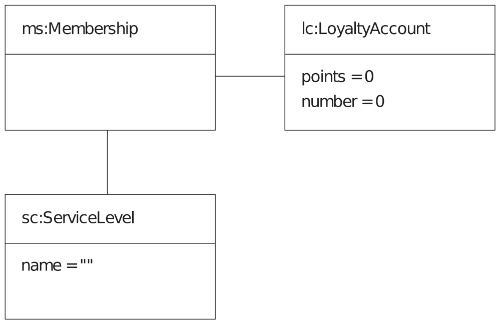
```
public void earn(Integer i){
   points += i;
   if(points > 100){
      membership.getCurrentLevel().setName("Gold");
   }else{
      if(points >= 0){
         membership.getCurrentLevel().setName("Silver");
      }else{
         membership.getCurrentLevel().setName("Inactive");
      }
   }
}
```

a) An implementation of the earn() method

```
public boolean post_earn(){
  String level = this.getMembership().getCurrentLevel().getName();
  boolean r0 = false;
  if ((!(this.getPoints() > 200) || level == "Platinum")
     && (!(this.getPoints() > 100 && this.getPoints() <= 200)
         || level == "Gold")
     && (!(this.getPoints() >= 0 && this.getPoints() <= 100)
         || level == "Silver")
     && (!(this.getPoints() < 0) || level == "Inactive")) {
       r0 = true; }
 return r0; }
```

(b) Java method generated from earn() OCL post-condition

Fig. 5. An implementation of the earn method and *post-condition*.



(a) Object diagram generated by code in (b)

```
public void test_case() {
    LoyaltyAccount lc = new LoyaltyAccount();
    Membership ms = new Membership();
    ServiceLevel sc = new ServiceLevel();
    lc.setMembership((ms);
    ms.setLoyaltyAccount(lc);
    ms.setServiceLevel(sc);
    sc.setMembership(ms);
    int i;
    i = 0;
    lc.earn(i); }
```

(b) Code to generate object diagram in (a)

Fig. 6. Test case values generation.

used interchangeably throughout the paper. Next, CCUJ calls method *earn()* on the test case value and employs concolic execution to guide the generation of further test case values. To tailor concolic execution for conformance checking, CCUJ tracks all object references, class fields, and method parameters derived from a class diagram. During symbolic execution, each statement updates the symbolic memory or the path condition based on previous symbolic values in the symbolic memory. The initial symbolic memory of *earn()* is obtained by executing the program in Fig. 6b and the path condition is initialized to true.

Next, we illustrate how CCUJ is used to reveal a software fault in method *earn()*.

- Trace I (Fig. 7):
  - The first execution based on the test case value, shown in Fig. 6b, starts with the execution of statement at line 2 at *earn* (Fig. 5a).
  - As a result, CCUJ updates the symbolic memory by creating a symbolic variable *$3* for parameter *i* and updating *$0.points* to *$0.points + $3* where *$0* denotes the heap object of *LoyaltyAccount* (row 1 in Table 1).
  - Next, the execution takes the *else* branch of the first *if* statement (line 3 of *earn()*) and the *then* branch of the second *if* statement (line 6 of *earn()*). Thus, the path condition for *earn()*, denoted by $pc_{earn()}$, is $\neg(\$0.points + \$3 > 100) \wedge \$0.points + \$3 >= 0$.
  - Next, CCUJ calls the *post-method post_earn()* and the *then* branch of the first *if* statement (line 4 of *post_earn()*) is taken. Thus, the path condition of *post_earn()*, denoted by $pc_{post\_earn()}$, is $\$0.points + \$3 <= 200 \wedge \$0.points + \$3 <= 100 \wedge \$0.points + \$3 >= 0 \wedge \$2.name = \text{“Silver”}$. Here, the collection of a path condition is based on the short-circuit evaluation employed by Java.

If method *post_earn()* returns false, then CCUJ reports that a software fault is found and its work is done. Otherwise, CCUJ calls a SAT solver, called Sat4j [10], to determine whether $pc_{earn()} \Rightarrow pc_{post\_earn()}$ is a tautology. If the implication relationship is a tautology, then all test case values satisfying $pc_{earn()}$ must satisfy $pc_{post\_earn()}$, and take the *same* path in *earn()* and *post_earn ()*, as shown in Fig. 7 (Trace I). Thus, all these test case values satisfying $pc_{earn()}$ all return *true*. Consequently, CCUJ looks for another test case value that does *not* satisfy $pc_{earn()}$ to change the execution path of method *earn ()*. To this end, CCUJ uses the backtracking technique to negate the sub-expressions in $pc_{earn()}$ by calling the SAT solver. In Trace I, *post_earn()* returns true and $pc_{earn()} \Rightarrow pc_{post\_earn()}$ is a tautology so CCUJ searches for another test case value as follows.

- Trace II (Fig. 7):
  - CCUJ calls the SAT solver to find a new test case value satisfying $\neg(\$0.points + \$3 > 100) \wedge \neg(\$0.points + \$3 >= 0)$, to enforce a different execution path. CCUJ uses a stack to store the path conditions collected during execution following a back-tracking approach. Thus, $\$0.points + \$3 >= 0$ is popped and flipped. In this case, the SAT solver returns an assignment that is used to generate the test value $\$0.points = 0$ and, $\$3 = -1$.
  - Next, CCUJ generates another simple object diagram with $\$0.points = 0$ *and* $\$3 = -1$, and uses Φ to produce a new test case.

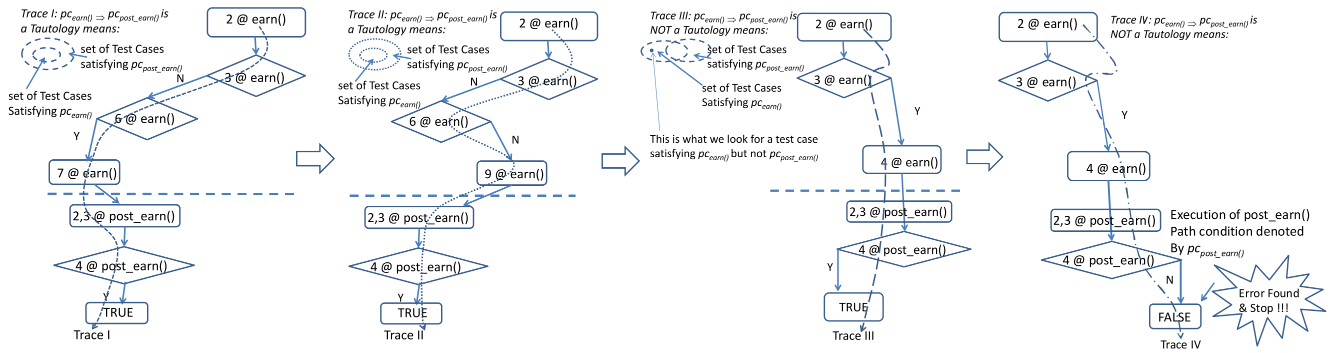

Trace I: $pc_{earn()} \Rightarrow pc_{post\_earn()}$ is a Tautology means:
set of Test Cases satisfying $pc_{post\_earn()}$
set of Test Cases Satisfying $pc_{earn()}$
2 @ earn()
3 @ earn()
N
6 @ earn()
Y
7 @ earn()
2,3 @ post_earn()
4 @ post_earn()
Y
TRUE
Trace I
Trace II: $pc_{earn()} \Rightarrow pc_{post\_earn()}$ is a Tautology means:
set of Test Cases satisfying $pc_{post\_earn()}$
set of Test Cases Satisfying $pc_{earn()}$
2 @ earn()
3 @ earn()
N
6 @ earn()
N
9 @ earn()
2,3 @ post_earn()
4 @ post_earn()
Y
TRUE
Trace II
Trace III: $pc_{earn()} \Rightarrow pc_{post\_earn()}$ is NOT a Tautology means:
set of Test Cases satisfying $pc_{post\_earn()}$
set of Test Cases Satisfying $pc_{earn()}$
This is what we look for a test case satisfying $pc_{earn()}$ but not $pc_{post\_earn()}$
2 @ earn()
3 @ earn()
Y
4 @ earn()
2,3 @ post_earn()
4 @ post_earn()
Y
TRUE
Trace III
Trace IV: $pc_{earn()} \Rightarrow pc_{post\_earn()}$ is NOT a Tautology means:
2 @ earn()
3 @ earn()
Y
4 @ earn()
2,3 @ post_earn()
Execution of post_earn() Path condition denoted By $pc_{post\_earn()}$
4 @ post_earn()
N
FALSE
Error Found & Stop !!!
Trace IV

Fig. 7. Different execution paths explored by CCUJ.

TABLE 1
Symbolic Memory and Path Condition after First Execution

Line No	Stmt	Symbolic Memory	Path Condition
2@earn ()	points+ = i	lc->$0; ms->$1;sc->$2;$0. membership->$1;$1.loyalty Account->$0;$1.serviceLevel- >$2;$2.membership->$1;this-> $0;i-> $3; $0.points->$0.points+$3	True
3@earn()	lf(points>100)	Same as the above	!($0.points +$3 > 100)
6@earn ()	if (points> = 0)	Same as the above	!($0.points +$3 > 100) and $0.points +$3 > = 0
7@earn ()	membership getCurrentLeval (). setName("Silver");	lc->$0; ms->$1;sc->$2;$0. membership->$1;$1.loyalty Account->$0;$1.serviceLevel- >$2;$2.membership->$1;this-> $0; i-> $3; $0.points->$0. points+$3; $2.name->"Silver"	Same as the above
2,2@post_earn()	String level = this.get Membership ().getCurrentLevel ().get Name (); r0 = false	lc->$0; ms->$1;sc->$2;$0. membership->$1;$1.loyalty Account->$0;$1.serviceLevel- >$2;$2.membership->$1;this-> $0; $0.points->$0.points+$3; $2. name->"Silver";level-> $4; r0- >$5;$5->false	True
4–7@post_earn()	lf(!(this.getPoints ()>200||. . .)	Same as the above	($0.points +$3) < = 200 and ($0.points +$3) < = 100 and ($0.points +$3) > = 0 and $2.name = "Silver"
8@post_earn()	return r0;	Same as the above	Same as the above

During this execution, CCUJ collects the two path conditions from the execution of *earn()* and *post_earn()*, i.e. $\neg(\$0.points + \$3 > 100) \wedge \$0.points + \$3 < 0$, denoted by $pc_{earn()}$, and $\$0.points + \$3 < = 200 \wedge \$0.points + \$3 < = 100 \wedge \$0.points + \$3 < 0 \wedge \$2.name = \text{"Inactive"}$, denoted by $pc_{post\_earn()}$ respectively.

Again *post_earn()* returns true and $pc_{earn()} \Rightarrow pc_{post\_earn()}$ is found to be a tautology by the SAT solver, as shown in Fig. 7 (Trace II). CCUJ tries to find another test case value to alter the execution path of *earn()* as follows.

- Trace III (Fig. 7):
  - CCUJ next flips the first sub-path condition to $\$0.points + \$3 > 100$ and sends it to the solver. The solver returns ($\$0.points = 0, \$3 = 150$), and CCUJ generates another set of test values, and calls method *earn()* again.
  - The two path conditions collected by CCUJ for *earn()* and *post_earn()* are $\$0.points + \$3 > 100$, denoted by $pc_{earn()}$, and $\$0.points + \$3 < = 200 \wedge \$0.points + \$3 > 100 \wedge \$0.points + \$3 > = 0 \wedge \$2.name = \text{"Gold"}$, denoted by $pc_{post\_earn()}$.

While *post_earn()* returns true, the SAT solver finds that $pc_{earn()} \Rightarrow pc_{post\_earn()}$ is not a tautology for Trace III, as shown in Fig. 7 (Trace III). Therefore some test case values that satisfy $pc_{earn()}$, following the same execution path of *earn()* in Trace III, do not follow the same execution path of *post_earn()* in Trace III. In this case, method *post_earn()* might return *false*. By negating one of the subconditions in $pc_{post\_earn()}$, a different execution path through *post_earn()* can be found. Thus, CCUJ attempts to find a test case value which alters the execution path of *post_earn()* as follows.

- Trace IV (Fig. 7):
  - CCUJ negates a sub-expression on $pc_{post\_earn()}$ using a back-tracking technique and sends the formula to the SAT solver which in this example returns ($\$0.points = 0, \$3 = 220$) and a new test case value is found and generated by CCUJ.
  - Finally, method *post_earn()* returns false on this test case, which means method *earn()* does not satisfy the *post-condition* defined in the class diagram. Therefore, a fault in Java implementation is found.

4 Definitions of Consistency Checking

In this section, we first formalize the consistency checking shown in Fig. 1. To do so, we introduce a set of functions related to a class diagram and a translation function Φ to model the generation of a skeletal code from a UML class diagram. These functions are related to the steps given in Figs. 1i, 1a, and 1ii. We further expand the translation function Φ to denote a translation of a Java method from an OCL constraint. Last, we give the definitions for consistency checking−shown at the bottom of Fig. 1—at three different levels.

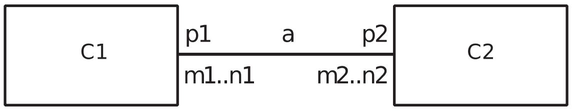


Fig. 8. An association and its properties.

An important input to consistency checking is a class diagram including OCL constraints. Thus, the domain of all class diagrams is represented by *ClassDiagrams.* The sets *Classes, Attributes, Operations, Associations,* and *Generalizations* represent all classes, attributes, operations, associations, and generalizations respectively. The function *classes: ClassDiagrams*→ $2^{Classes}$ maps a class diagram to all the classes in the diagram. The function *attributes:* $Classes \rightarrow 2^{Attributes}$ maps a class to its all properties/attributes. The function *operations:* $Classes \rightarrow 2^{Operations}$ maps a class to its all operations/methods. The function *associations:* $Classes \rightarrow 2^{Associations}$ maps a class to all the associations whose ends include the class itself. For instance, in Fig. 8, *associations(C1)* = *{a}* since association *a* is the only one connected to class *C1.* The set *P* is a set of owned association ends derived from all associations. In [2], an association end owned by a class is also an attribute; thus, $P \subseteq$ *Attributes.* The function *props: Associations* $\rightarrow P \times P$ maps a binary association[1] to its two properties. Also, the function *minMul:* $P \rightarrow \mathbb{N}\ U\{0\}$ maps each property to the minimal multiplicity value specified for the property. The functions *source: P* → *Classes* and *target: P* → *Classes* map each property to its source end and target end respectively. For instance, in Fig. 8, *props(a)* = *{p1,p2}, target (p1)* = *source(p2)* = *C2, target(p2)* = *source(p1)* = *C1, minMul (p1)* = *m1, and minMul(p2)* = *m2.*

The constraints written in OCL include *pre-conditions* and *post-conditions* of operations in a class as well as class invariants. A class invariant is converted to a *pre-condition* and *post-condition* of *all* methods in the class. The domain of all OCL constraints is denoted as *OCLConstraints*. The function *OP2OCL_PRE: Operations* → *OCLConstraints* maps each operation to its *pre-condition* and the function *OP2OCL_POST: Operations* → *OCLConstraints* maps each operation to its *post-condition*.

We introduce a translation function, also called a translation schema, denoted as Φ, which consists of two parts. The first part models the translation from a class diagram denoted as *CD* into a Java program denoted as P_{CD} as produced by a forward engineering tool shown in Fig. 1a. Thus, we denote the translation schema as $\Phi : Class\ Diagrams \rightarrow Java\ Programs$, where *JavaPrograms* represents all Java programs. As mentioned before, each generated Java class should have a constructor and setter/getter methods for each attribute. As an example, a translation schema used by most forward engineering tools is given as follows:

1. Each UML Class *C* is mapped to a Java class *JC* with the same name, denoted by *JC* = Φ (C). The constructors have the implementation details given by Φ.
2. Each property is mapped to a class field with setter and getter methods, and both property and class fields have the same name while their setter and getter method names are prefixed with *set* and *get* respectively. These setter and getter methods have the implementation details given by Φ.
3. Each UML operation *op* is mapped to a Java method *m* with the same signature such as name, denoted by $m = \Phi(op)$, but the implementation details are skipped.
4. Each association is treated as a property so the translation schema used for properties is applied to associations. Also, each inheritance is implemented by keyword *"extends"* in Java. If a multiple-inheritance occurs, the translation schema Φ is tool-dependent. For example, some tools keep one as an inheritance and convert each of the others to an association relationship.

The second part of the translation schema translates each OCL constraint into a Java Boolean method and insert it into the appropriate class, to ensure that a final Java implementation satisfies all OCL constraints given in a class diagram. Likewise, the translation schema described in the OCL book by Warmer and Kleppe (Chapter 4) [3] is an example of the second part of Φ. Java methods that are converted from a *pre-*, and *post-condition* of an operation are called Java *pre-method* and *post-method* respectively. Specifically, for each method *m* implementing an operation *op* in a class diagram, we can get the *pre-condition* of *op* via *OP2OCL_PRE(op),* and the *post-condition* via *OP2OCL_POST(op)*. By applying Φ, we get a *pre-method* and *post-method* of *m,* denoted by m_{pre} = *F(OP2OCL_PRE(op))* and m_{post} = *F(OP2OCL_POST(op))*, respectively.

To define consistency checking, we employ the operational semantics to model execution of a Java program. In brief, we introduce a state function *s*, which maps a heap object and its field to a concrete value, to denote a status of a heap object at a given state *s*. All states where all heap objects can be generated via the first part of Φ are denoted as S_Φ. Remember that Φ denotes implementation for constructors, setter and getter methods. A test case value is an element of S_Φ. Next, we use $<e, s>$ to denote a concrete configuration and $<e, s> \rightarrow <e', s'>$, called a reduction rule in the operational semantics, models a single reduction step of expression *e* in state *s* and the outcome of the evaluation is a reduced expression e' in a new state s'. In particular, if $\alpha.m\ (\bar{v})$ denotes a method call *m* on object α where $\bar{v}$ is a list of arguments, then we use $<\alpha.m(\bar{v}), s> \rightarrow^* <e', s'>$ to model the execution of method $\alpha.m\ (\bar{v})$ in state *s* and $\rightarrow^*$ denotes the transitive and reflexive closure of →. The outcome of the execution is a reduced expression e' in a new state s'. Note that we translate OCL constraints into Java Boolean methods inserted into the corresponding Java classes, and OCL is a side-effect free language. Thus, the translated Java methods do not have any object creation statement and assignment statement to change the status of all heap objects. Namely, the execution of any Boolean method translated from an OCL constraint should not change the state from which the method starts with, i.e. $<\alpha \cdot \Phi(op), s> \ \rightarrow^* <true/false, s>$. Next, we define consistency checking via the conformance relationship at different levels.

We first consider the method conformance, i.e., the consistency checking between an operation of a UML Class and

1. CCUJ only considers binary associations.

its corresponding Java implementation. Informally speaking, a UML operation *op* is mapped to a void method m via the translation function Φ. The goal of CCUJ is to find whether the implementation of the method body satisfies all constraints related to *op* by checking the corresponding Java pre-methods and post-methods. Following method conformance, we define the consistency checking between a UML class and its Java class, i.e., class conformance and finally between a UML class diagram and its (final) Java implementation, i.e., implementation conformance.

Definition 1. *A state* s $\in S_\Phi$ *satisfies an OCL constraint* c *of object* α, *denoted as* $s \models_{(\Phi,\alpha)} c$, *if and only if* $<\alpha.\Phi(c), s> \rightarrow^* <true, s>$.

Definition 2 (Method conformance). *A Java method* m *of object* α *conforms to its corresponding operation* op *in a class diagram with respect to* Φ, *denoted by* $m \sqsubseteq_\Phi op$, *if* $\forall s \in S_\Phi(s \models_{(\Phi,\alpha)} OP2OCL\_PRE(op) \Longrightarrow (\exists s' \in S_\Phi(<\alpha.m(\bar{v}), s> \rightarrow^* <e', s'>) \Longrightarrow (s' \models_{(\Phi,\alpha)} OP2OCL\_POST(op))))$ *where* $\Longrightarrow$ *is the logical "implies" operation. Otherwise, the method in the Java class does not conform to its UML counterpart with respect to* Φ, *denoted by* $m \not\sqsubseteq_\Phi op$.

Definition 3 (Class conformance). *A Java class* JC *conforms to its UML class* C *with respect to* Φ, *denoted by* $JC \sqsubseteq_\Phi C$ *if* $\forall\ op \in operations(C), \exists\ !\ m(m = \Phi(op) \Longrightarrow m \sqsubseteq_\Phi op)$. *Otherwise, the Java class* JC *does not conform to its UML class* C *with respect to* Φ, *denoted by* $JC \not\sqsubseteq_\Phi C$.

Definition 4 (Implementation conformance). *A Java implementation* I *conforms to its UML class diagram* D *with respect to* Φ *if* $\forall C \in classes(\mathrm{D}), \exists\ !\ JC(JC = \Phi(C) \Longrightarrow JC \sqsubseteq_\Phi C)$. *Otherwise, the Java implementation* I *does not conform to its UML class diagram* D *with respect to* Φ, *denoted by* $I \not\sqsubseteq_\Phi D$. *For brevity, we skip "with respect to* Φ*" throughout the paper since the translation schema* Φ *is given.*

5 Algorithm in CCUJ and Its Rigorous Analysis

In Section 5.1, we describe the main algorithms in CCUJ, which is the implementation issue of the bottom of Fig. 1. Section 5.2 provides the rigorous analysis about the correctness of the CCUJ algorithm. To simplify the discussion, the CCUJ algorithm and the rigorous analysis are discussed only on method conformance as defined in the previous section. Class conformance and implementation conformance can be extended from method conformance. In order to show the rigorous analysis, we introduce a simple version of Java to obtain a small calculus, for which the rigorous analysis is not only possible but also easy. Using a reduced language to make proof possible and easy was first presented by [11] and followed by many researchers [12], [13], [14]. The syntax of the simple Java-like language only includes the Boolean type and all classes as types in the language. More details about the simple Java-like language with its both concrete and concolic operational semantics can be found in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2015.2488645.

A distinguishing feature of CCUJ is that it efficiently prunes test cases by investigation of the logical *implies* relationship between the symbolic path constraints for the method under test and its Java post-method. For the rigorous analysis, we show that if CCUJ returns *true* then a method under test correctly implements its post-method (soundness); and if a method under test correctly implements its post-method, then CCUJ returns *true* (completeness).

```
Configuration concolic_exe(conf, stack_pc)
{
    current_conf ⟵ conf;
    while (there exists a rule r: ($e_1$,$\omega_1$)⟶ ($e_2$,$\omega_2$) applicable to current_conf)
    {
        if($e^l$ is added to $\omega_2$.C in r)
        {   stack_pc.push($e^l$);
            if(r  is one of SR-Read-Field, SR-Field-Assignment, SR-Method-Call)
              stack_pc.[top].flipped=true;
        }
        current_conf ⟵ update current_conf based on ($e_1$,$\omega_1$)⟶ ($e_2$,$\omega_2$);
    }
    return current_conf;
}
```

Fig. 9. The algorithm of concolic execution.

5.1 Algorithm of CCUJ

In this section, we concentrate on the main algorithm of CCUJ which enforces method conformance given by Definition 2, the heart of consistency checking. Class conformance can be achieved if all methods in a class satisfy method conformance. Likewise, implementation conformance is satisfied if all classes in a Java implementation satisfy class conformance. Since CCUJ employs concolic execution to investigate method conformance, we introduce some important concepts related to concolic execution. A concolic configuration or simply called a configuration used in a concolic execution, is denoted as a tuple $<e, \boldsymbol{\omega}>$ where *e* is a Java expression to be evaluated in concolic state $\boldsymbol{\omega}$. A concolic state $\boldsymbol{\omega}=<\boldsymbol{\omega}.\boldsymbol{\gamma},\boldsymbol{\omega}.\Gamma,\boldsymbol{\omega}.\mathrm{C}>$ consists of three parts, denoted by $\boldsymbol{\omega}.\boldsymbol{\gamma}, \boldsymbol{\omega}.\Gamma$, and $\boldsymbol{\omega}.\mathrm{C}$ respectively. $\boldsymbol{\omega}.\boldsymbol{\gamma}$, represents a concrete state, and $\boldsymbol{\omega}.\Gamma$, represents a symbolic state. Similar to a concrete state, a concolic state maps a heap object and its field to a symbolic value. A path constraint consisting of all path conditions related to $\boldsymbol{\omega}$ is denoted by $\boldsymbol{\omega}.\mathrm{C}$. Similar to concrete execution, the operational semantics of concolic execution is given by a set of reduction rules such as $<e, \boldsymbol{\omega}> \rightarrow <e', \boldsymbol{\omega}'>$. A path constraint C given by $\boldsymbol{\omega}'.C$ consists of an ordered set of $e_1^{(l_1,t_1)} \ldots e_i^{(l_i,t_i)} \ldots e_n^{(l_n,t_n)}$where $l_i \in$ *LabelNames*, which is a label of a condition statement, *and* t_i is either *true* or *false* where $i \in \{1, \ldots, n\}$, and n is the total number of branch expressions each of which is called a path condition denoted as pc_i, i.e. $pc_i = e_i^{(l_i,t_i)}$.

Before introduction of the main algorithm, we discuss three helper methods: *Configuration concolic_exe(conf, stack_pc)*, *C backtracking(stack_pc, Pre)* and *Stack clean_stack (stack_pc,list)* where *Configuration*, *C*, and *Stack* denote a configuration, a path constraint, and the stack data structure respectively. The method *concolic_exe(conf, stack_pc)*, shown in Fig. 9, concolically executes a concolic configuration *conf* using the stack *stack_pc* containing all relevant path conditions. Initially, the method assigns conf to the current configuration denoted as *current_conf*. Next, the method checks whether a reduction rule r can be applied to current_conf. Note, since the simple Java-like language is deterministic, there is only one reduction rule which can be found. If yes, the method checks whether a path condition should be

```
C backtracking(stack_pc, Pre)
{
    C result ← false;
    while(top>0 && (stack_pc[top].flipped ||
            stack_pc[top].flipped==false && ! Satisfiable(stack_pc[1]∧..∧!stack_pc[top]∧Pre)))
     {    stack_pc[top].flipped=false;     stack_pc.pop(); }

     if(top>0)
     {  stack_pc[top].flipped=true;
        stack_pc[top] = !stack_pc[top];
        result ← stack_pc[1]∧..∧stack_pc[top];
        stack_pc ← null;
     }
    return result;
}
Stack clean_stack(stack_pc, list)
{
     while(stack_pc[top]∈ list)
     {    stack_pc[top].flipped=false;                  stack_pc.pop(); }
      return stack_pc;
}
```

Fig. 10. The algorithms of back-tracking and cleaning a stack.

added to the C component of the target configuration based on the reduction rule. If yes, the path condition is pushed onto the stack *stack_pc*. Furthermore, if the path condition is one of the conditions added based on SR-Read-Field, SR-Field-Assignment, and SR-Method-Call, then the flipped value of the path condition should set to *true* since they are not flippable to void a runtime error. Then, the current configuration is updated based on the reduction rule. This procedure continues until no reduction rules can be found; and the method finally returns the current configuration.

The method *backtracking(stack_pc, Pre)*, shown in Fig. 10, aims to find a path constraint via backtracking such that the returned constraint is satisfiable. The returned constraint consists of the conjunction of parameter *Pre* and some but not all path conditions given by the parameter *stack_pc*. To this end, the method needs to find a top path condition on stack *stack_pc* such that the path condition has not yet been flipped and construct a new path condition of the form *stack_pc*[1] ∧..∧ *stack_pc[top*-1*]* ∧ *!stack_pc[top]* ∧ *Pre* that is satisfiable. *!stack_pc[top]* denotes the negation of the path condition given by *stack_pc[top].* To find such a path condition, the method pops the top element off *stack_pc* if the top element, i.e., *stack_pc[top]*, has been flipped before or *stack_pc*[1] ∧..∧ *stack_pc[top*-1*]* ∧ *!stack_pc[top]* ∧ *Pre* is not satisfiable. When a top element is popped up, the method resets the flipped value to false since the popped path condition can be flipped in the future path search. If the path constraint is found, then method sets the *flipped* attribute for the top element on *stack_pc* to *true*, which means this path condition has been flipped and will not be flipped again when expanding path conditions *stack_pc[1].. stack_pc[top-1]*. Next, the method dumps all path conditions on the stack to *result* and sets the stack to be empty. Finally, the method returns the *result* path condition. The method *clean_stack (stack_pc,list)*, shown in Fig. 10, removes the path conditions in *list* from *stack_pc*. This method is used to exclude certain path conditions when finding the next test case. Specially, the excluded path conditions are not flappable in the search of the next test case value.

Fig. 11 shows how the algorithm Concolic_CCUJ of CCUJ tests a method under test against its post-method by integrating symbolic execution into concrete execution. The input to the algorithm includes the object α with its method *m* to be tested, the post-method m_{post} translated from the OCL post-condition, the path-condition *Pre* collected from the OCL pre-condition.[2] During the execution of algorithm Concolic_CCUJ, we also introduce local variables *S*, **B**, *stack_pc*, and *new_pc*. Variable *S* is used to store a path constraint including *Pre* to find a test case value for one concolic execution within the while statement. Variable **B** stores all path constraints that have been executed by Concolic_CCUJ. Variable *stack_pc* denotes a stack structure which stores all path conditions during one concolic execution within the while statement while *new_pc* holds the path constraint returned by method *backtracking*.

The algorithm includes a main while loop and each iteration of the loop collects an execution path, denoted by a path constraint, of methods *m* and m_{post} via a symbolic execution combined with concrete execution. The path constraint is used to prune all the states that have the same execution path as the current execution. More specifically, *S* is initialized by the *Pre* condition (at line 1), which denotes an initiate heap satisfying the pre-condition. If *S* is not satisfiable (at line 2), then no test case value/state satisfies *Pre* and the algorithm returns *true.* Otherwise, a state satisfies the pre-condition via an assignment and CCUJ constructs an initial concolic state $\boldsymbol{\omega}$ whose concrete part $\boldsymbol{\omega}.\boldsymbol{\gamma}$ satisfies *S (*at line 6). CCUJ then concolically executes $<\alpha.\mathrm{m},\ <\boldsymbol{\omega}.\boldsymbol{\gamma}, \boldsymbol{\omega}.\Gamma, \{\}>>$ at line 6 via calling method *concolic_exe(*$<\alpha.\mathrm{m}, <\boldsymbol{\omega}.\boldsymbol{\gamma}, \boldsymbol{\omega}.\Gamma, \{\}>>$*, stack_pc)*. During a concrete execution, a run time error occurs such as when a method is called on a null object and in this case an error configuration, denoted as $\perp$, occurs according to the operation semantics of concrete execution. Accordingly, the operational semantics of concolic execution results in an error symbolic configuration. If an error configuration is produced in method *m* at line 7, then CCUJ stops and returns

2. The maximum loop execution count k is preset by CCUJ.

```
Algorithm Concolic_CCUJ(α, m, $m_{post}$, Pre)
REQUIRES: α is an object whose method m is to be checked,
          m is a method to be checked,
          $m_{post}$ is a post-method of m,
          Pre is a formula representing the precondition of m
OUTPUTs: true/false
1.  S ← Pre, B ← null, stack_pc ← null, new_pc ← null
2.  if(S is not satisfiable)
3.    return true;
4.  else
5.    while (S is satisfiable)
6.      $<e', \boldsymbol{\omega}_m>$ ←concolic_exe(< α .m, $< \boldsymbol{\omega}.\boldsymbol{\gamma}, \boldsymbol{\omega}.\boldsymbol{\Gamma}, \{\}>>$,stack_pc) where $\boldsymbol{\omega}.\boldsymbol{\gamma}$ satisfies S
7.      if $< e', \boldsymbol{\omega}_m >$ is an error configuration then
8.          return false
9.      $<e'', \boldsymbol{\omega}_{post} >$← concolic_exe(< α .$m_{post}$, $< \boldsymbol{\omega}_m.\boldsymbol{\gamma}, \boldsymbol{\omega}_m.\boldsymbol{\Gamma}, \{\}>>$,stack_pc)
10.     if $e'' == false$
11.         return false  // "bug is found"
12.     else
13.         add $\boldsymbol{\omega}_m$.C + $\boldsymbol{\omega}_{post}$.C to B
14.         if $\boldsymbol{\omega}_m$.C ⇒ $\boldsymbol{\omega}_{post}$.C is not tautology then
15.             new_pc ← backtracking(stack_pc, $Pre$)
16.         else // $\boldsymbol{\omega}_m$.C ⇒ $\boldsymbol{\omega}_{post}$.C is tautology
17.             stack_pc ⟵ clean_stack(stack_pc, $\boldsymbol{\omega}_{post}$.C)
18.             new_pc← backtracking(stack_pc, $Pre$)
19.         S ← Pre ∧ new_pc
20. return true;
```

Fig. 11. Algorithm of CCUJ.

false at line 8. Otherwise, CCUJ concolically executes the post-method m_{post} based on the state produced by the execution of method *m* at line 9. Here, CCUJ resets $\boldsymbol{\omega}_m$.C to an empty set. Thus, the C component in the final concolic state includes only path conditions collected during the concolic execution of m_{post}. Similarly, CCUJ checks the return value of m_{post} and if the return value is *false* at line 10, an error configuration is found and CCUJ stops and returns *false*. Otherwise, CCUJ finds a new path constraint based on the current concolic execution and adds the path constraint to **B** at line 13. Next, CCUJ further analyzes the two path conditions of methods *m* and m_{post}, which are $\boldsymbol{\omega}_m.\mathrm{C}$ and $\boldsymbol{\omega}_{post}.\mathrm{C}$ respectively, as follows.

- If $\boldsymbol{\omega}_m.\mathrm{C} \Longrightarrow \boldsymbol{\omega}_{post}.\mathrm{C}$ is not a tautology, then CCUJ searches for a new state with the same execution path of *m* but a different execution path of m_{post} at line 15 by calling method *backtracking(stack_pc,Pre)*. Note stack *stack_pc* contains all path conditions collected from the execution of methods *m* and m_{post}.
- If $\boldsymbol{\omega}_m.\mathrm{C} \Longrightarrow \boldsymbol{\omega}_{post}.\mathrm{C}$ is a tautology, then CCUJ searches for a new state with a different execution path of *m* at lines 16, 17 and 18. Since all test case values satisfying pc_m also satisfy pc_{post}, CCUJ calls method *clean_stack (stack_pc,* ω_{post}*.C)* to remove all path conditions generated during m_{post} from *stack_pc*. Thus, stack *stack_pc* only includes the path conditions from method *m*. Next, CCUJ tries to find a path constraint to change the execution path for method *m* by calling method *backtracking(stack_pc,Pre)*.

To simplify the consistency checking between a Java program and its class diagram, algorithm *Concolic_CCUJ* only considers the path condition *Pre* which is extracted from the pre-method m_{pre} translated from an OCL pre-condition. The simplification leverages the following rigorous analysis about the consistency checking in CCUJ without loss of generality. In fact, we can collect the path condition *Pre* by calling m_{pre} on α. Similar to *Concolic_CCUJ*, both the concrete and symbolic executions of m_{pre} are carried out. If one execution returns, we alter the path condition to find another execution path. This procedure is not stopped until all execution paths in m_{pre} are explored. If one execution returns true, we attach the collected path conditions to the ones in *Pre* via logical conjunction, i.e., *and*. We drop the path conditions for the executions returning false.

5.2 Rigorous Analysis of the Algorithm

Next, we provide rigorous analysis of the correctness of the algorithm in Fig. 11. Note that Definitions 2 is based on (the semantics of) the concrete execution of a Java program while the algorithm in Fig. 11 uses concolic execution. Consequently, to provide the rigorous analysis of the algorithm, we show that Definition 2, can be obtained using (the semantics of) the concolic execution. The proofs of all lemmas and theorems given in this section can be found in the Appendix, available in the online supplemental material.

As a starting point, we consider some properties related to a single concolic reduction step in Lemma 1. For expression *e* and state *s*, assume ω_s is the corresponding concolic state, and there is a reduction rule related to expression *e* and ω_s, i.e. $< e, \boldsymbol{\omega}_s > \rightarrow < e', \boldsymbol{\omega}'_s >$. First, if the target concolic configuration is an error configuration then the corresponding target concrete configuration is also an error configuration. Second, all reduction rules given in concolic execution maintain the path conditions' satisfaction during execution from a source configuration to a target configuration. Third, contrary to the second statement, if a state satisfies the path conditions of a target configuration, then the state must satisfy the path conditions of the source configuration. Last, for any state σ such that σ satisfies the path condition produced by the reduction rule, i.e., $\boldsymbol{\omega}'_s.C$, we ensure that the corresponding concrete reduction step can be represented by the concolic reduction step and that the final path constraint produced by concolic state ω_σ is the same as the path constraint produced by ω_s.

Lemma 1. *For any state* s, $\omega_s = <\omega_s.\boldsymbol{\gamma}, \omega.\Gamma, \omega_s.C>$, *is the corresponding concolic state, and e is an expression, if* $<e, \omega_s> \rightarrow <e', \boldsymbol{\omega'_s}>$ *then*

i) *If* $<e', \boldsymbol{\omega}'_s>$, *is an error configuration* $\perp$ *in concolic execution, then* $<s(e'), s(\boldsymbol{\omega}'_s.\Gamma)>$ *is an error configuration* $\perp$ *in the concrete execution.*
ii) *If* $\omega_s.C$ *is satisfied by s then* $\boldsymbol{\omega}'_s.C$ *is satisfied by s.*
iii) *For any state* σ, *if* $\boldsymbol{\omega}'_s.C$ *is satisfied by s, then* $\omega_s.C$ *is satisfied by* σ.
iv) *For any state* σ *whose concolic state is* $\omega_\sigma = <\omega_\sigma.\boldsymbol{\gamma}, \omega.\Gamma, \omega_\sigma.C>$ *such that* $\omega_\sigma.C = \omega_s.C$ *and* $<e, \omega_\sigma> \rightarrow^* <e', \omega'_\sigma>$, *if* $\boldsymbol{\omega'_s}.C$ *is satisfied by* σ, *then 1) we have* $<\sigma(e), \sigma> \rightarrow^* <\sigma(e'), \sigma(\boldsymbol{\omega'_s}.\Gamma)>$; *and 2)* $\boldsymbol{\omega'_\sigma}.C = \boldsymbol{\omega'_s}.C$.

The proof is based on induction on the structure of *e*.

Next, we can consider the properties related to multiple reduction steps due to a complex structure of expression *e*. Similar to the single reduction step case, we have four properties of interest. The meaning of the first two properties is obvious. The third property states that all initial states that satisfy the path condition produced by ω_s, i.e $\boldsymbol{\omega}'_s.C$, should have the same execution path as state *s*. Last property ensures that for any state σ such that σ satisfies $\boldsymbol{\omega}'_s.C$, the corresponding multiple concrete reduction steps for state σ can be represented by the corresponding concolic reduction steps for ω_s.

Lemma 2. *For any state* s *whose symbolic state is* $\omega_s = <\omega_s.\boldsymbol{\gamma},\omega.\Gamma, \{\}>$, *if* $<e, \omega_s> \rightarrow^* <e', \boldsymbol{\omega}'_s>$ *then*

i) *If* $<e', \boldsymbol{\omega}'_s>$ *is an error concolic configuration* $\perp$, *then* $<s(s(e'), s(\omega'_s.\Gamma)>$ *is an error configuration* $\perp$ *in concrete execution.*
ii) *State s satisfies* $\boldsymbol{\omega}'_s$*.C.*
iii) *For state* σ, *if* σ *satisfies* $\boldsymbol{\omega}'_s$.C *and* $\omega_\sigma = <\omega_\sigma.\boldsymbol{\gamma}, \omega.\Gamma, \{\}>$ *is the corresponding concolic state, then* $\boldsymbol{\omega}'_s$.C = $\boldsymbol{\omega}'_\sigma$.C *where* $< e, \omega_\sigma > \rightarrow^* <e', \omega'_\sigma>$.
iv) *For state* σ, *if* σ *satisfies* $\boldsymbol{\omega}'_s$*.C, then we have* $<\sigma(e), \sigma > \rightarrow^* <\sigma(e'), \sigma(\omega'_s.\Gamma)>$.

The proof is based on induction on the length of the symbolic transitions.

Now, we show that all execution paths in the form of path constraints in expression $\alpha.m; \alpha.m_{post}$ are stored in variable **B** based on the algorithm shown in Fig. 11. Note $\alpha.m; \alpha.m_{post}$ has a limited number of sub-expressions and we set *k* as a bound for the number of executions for a while statement. The next theorem shows variable **B** has a finite number of path constraints and includes all path constraints of $\alpha.m; \alpha.m_{post}$.

Theorem 1. *Assume at the end of the execution of* Concolic_CCUJ *on the expression* $\alpha.m; \alpha.m_{post}$, *the set* **B** *satisfies the following properties:*

i **B** *contains the finite number of path constraints.*
ii **B** *contains all path constraints in* $\alpha.m; \alpha.m_{post}$.

One important feature of CCUJ's algorithm is the study of the path constraints produced by $\alpha.m$ and $\alpha.m_{post}$ to prune some test case values. The following theorem claims the soundness of the pruning technique used in CCUJ.

Theorem 2. *For any state* **s** *whose symbolic state is* $\omega_s = <\omega_s.\boldsymbol{\gamma}, \omega.\Gamma, \{\}>$, *assume* $<e_1; e_2, \omega_s> \rightarrow^* <e_2, \omega_{s_1}> \rightarrow^* <e, \omega_{s_2}>$ *and let* pc_1 *be* $\omega_{s_1}.C$ *and* pc_2 *be* $\omega_{s_2}.C - \omega_{s_1}.C$ *where* $\omega_{s_2}.C - \omega_{s_1}.C$ *denotes to remove all path conditions of* $\omega_{s_1}.C$ *from* $\omega_{s_2}.C$. *For any state* σ *whose concolic state is* $\omega_\sigma = <\omega_\sigma.\boldsymbol{\gamma}, \omega.\Gamma, \{\}>$ *such that* σ *satisfies* $\omega_{s_1}.C$ *and* $<e_1; e_2, \omega_\sigma> \rightarrow^* <e_2, \omega_{\sigma_1}> \rightarrow^* <e, \omega_{\sigma_2}>$ *if* $\mathrm{pc}_1 \Longrightarrow \mathrm{pc}_2$ *is a tautology, then* $\omega_{s_2}.C = \omega_{\sigma_2}.C$.

Next, we consider the correctness of the algorithm in terms of soundness and completeness. Intuitively, the soundness of the algorithm means that if the algorithm returns *true* then a method under test correctly implements its post-method. Conversely, the completeness of the algorithm means if a method under test correctly implements its post-method, then the algorithm returns *true*. To demonstrate this property, we first assume the algorithm terminates as follows.

Theorem 3. *Assume* Concolic_CCUJ *terminates.* Concolic_CCUJ *returns true if and only if for any state* s, *if* s *satisfies* Pre, *then* $(<\alpha.m, s> \rightarrow^* <e', s_m>) \wedge (<\alpha.m_{post}, s_m> \rightarrow^* <true, s_{post}>)$ *is true.*

Next theorem claims that the correctness of *Concolic_CCUJ* based on the method conformance, Definition 2, is ensured when *Pre, α.m,* and $\alpha.m_{post}$ are set appropriately based on a class diagram and its Java implementation.

Corollary 1. *Assume operation* op *in class* C *is implemented by method* m, *namely* m = Φ(op). Pre *is an equivalent constraint of* Φ(OP2OCL_PRE(op)), *and* m_{post} *is the post-method translated from the post-condition via* Φ, *namely* m_{post} = Φ(OP2OCL_POST(op)). *Assume* Concolic_CCUJ *terminates.* Concolic_CCUJ *returns true if and only if* $m \sqsubseteq_\Phi$ *op.*

The proof of the above corollary is obvious based on Theorem 3. Likewise, we have the corresponding corollaries related to the class conformance and implementation conformance if we consider all methods in a class and all classes in a Java implementation respectively.

Last, the following theorem shows that the algorithm terminates.

Theorem 4. *Algorithm* Concolic_CCUJ *terminates.*

To prove theorem 4, note that, during each execution of the while statement in *Concolic_CCUJ*, there is only one new path constraint added to **B.** Since the number of path constraints in **B** is finite according to theorem 1, the execution of the while statement stops. Therefore, algorithm *Concolic_CCUJ* terminates.

6 Implementation of CCUJ

In this section, we will discuss some implementation issues related to CCUJ. We first introduce how an OCL constraint as a post-condition is converted to a Java post method (Section 6.1). Next, we present some implementation issues related to algorithm *Concolic_CCUJ* such as the generation of test cases, symbolic execution, and black box methods (Sections 6.2, 6.3, and 6.4 respectively). Finally, we discuss the limitation of CCUJ (Section 6.5).

6.1 Preliminary Testing Preparation

The initial step on the preliminary testing preparation consists of collecting information from a class diagram that will be used to generate test cases during the later dynamic testing phase and the OCL constraints translation. To parse a class diagram, CCUJ uses the MDT EMF [15] implementation, part of the Eclipse project. The information extracted while parsing the class diagram includes class names, associations, fields, and inheritance. This information is saved on a data structure for access during the dynamic testing step.

The next step consists of parsing OCL constraints from the class diagram and translating them into Boolean Java methods. To parse an OCL constraint, CCUJ employs the MDT OCL implementation [16], also a part of the Eclipse project, to create a Java Boolean method. To implement the translation of OCL constraints to Java code, CCUJ follows the templates and guidelines introduced by Warmer et al. [3], and the implementation is based on the syntax given in Appendix B, available in the online supplemental material, of [3]. More specifically, CCUJ initially employs the MDT OCL to generate an Abstract Syntax Tree (AST) for each OCL constraint. A visitor pattern is used to visit children nodes of interests. For the non-leaf nodes, the *visit()* method further traverses the child node(s) by calling method *accept ()* in the Visitor pattern. Each *visit()* method produces a partial Java code based on the OCL expression represented by a node and returns it to its parent node. In addition, method *visit()* maintains local variables so the value of each sub-expression can be properly stored. Finally, a root node collects all code fragments to produce the final complete Java code for a *post-method*.

6.2 Dynamic Testing: Test Case Generation

As aforementioned, each test case is actually generated by the execution of a program built based on the translation schema Φ. In order to simplify the consistency checking between a Java implementation program and its class diagram, CCUJ uses only multiplicities from the class diagram rather than the OCL constraints as pre-conditions. These multiplicities are converted to a formula and sent to a SAT solver to create a test case. Thus, as an initial step during dynamic testing, CCUJ generates a minimum test case. While any state where the number of objects satisfies the multiplicities can be used as a test case, the state with the smallest number of objects is always the easiest one generated using the constructors, and setter methods. Furthermore, the smallest number of objects considered by the initial test case does not affect the results works since algorithm *Concolic_CCUJ* explores all paths by excluding all previously explored paths from each new test case.

To guarantee that the SAT solver only produces valid assignments in terms of multiplicity, CCUJ constructs the SAT solver's class field domains in such a way that no value from the domain can violate the multiplicity constraints. A domain, in the context of a SAT solver, is the set of allowable values that can be assigned to a particular variable. To

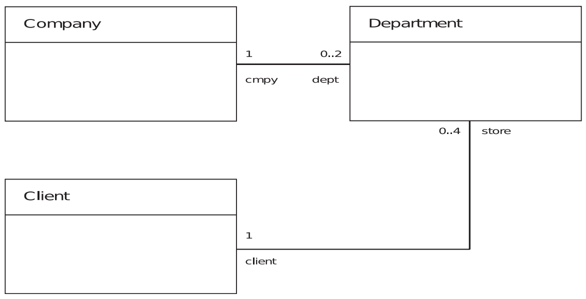


Fig. 12. Company class diagram.

construct the domains, CCUJ first parses the path condition to select all the variables that correspond to an association end in the class diagram. Next, for each class field, CCUJ gets its lower bound multiplicity, l, and upper bound multiplicity, h. Based on l and h, the class field domain is defined such that for each *value* in the domain, $value >= l$ and $value <= h$. If the upper bound is unlimited, CCUJ generates up to c values, where c is a constant. To illustrate this approach, consider the class diagram in Fig. 12 and the path condition $cmpy.dept\ != null$, where *cmpy* is an instance of class *Company* and *dept* is the association end *dept* in the association between *Company* and *Department*. Based on the multiplicity of 0..2, CCUJ generates a domain with the values [0, 1, 2], where 0 represents the *null* value, and 1 and 2 indicate that the class field can be assigned with one or at most two instances of type *Department*. For class fields that do not implement an association end, CCUJ generates a predefined number of values depending on the variable type.

In some instances, multiple class fields implementing association ends may share the same class but have different multiplicity values. In this case, CCUJ takes the largest multiplicity values to instantiate the number of object of this class. For example, the two association ends *dept* and *store* in Fig. 12 are connected to Class *Department* but have the multiplicities 0..2 and 0..4 respectively. Thus, the number of objects instantiated from Class *Department* is four, and CCUJ uses domain [0, 1, 2, 3, 4]. However, the values in the domainthe number of objects instantiatedmay not be enough to enforce the multiplicity constraints for all class fields implementing the association ends. In this case, the field in Class *Company* implementing the association end *dept* can have at most two objects of *Department.* CCUJ adds one or more sub-expressions to the formula sent to the SAT solver in order to enforce all the constraints. Namely, to enforce the multiplicity constraint for the class field *dept*, CCUJ adds the sub-expression $cpmy.dept <= 2$ to the formula sent to the SAT solver.

After the initial construction of a domain, CCUJ also dynamically updates the domain values, if necessary, using concrete values and path conditions encountered during a previous test. The domain update is performed to prevent trivially unsatisfiable formulas when flipping branch conditions, due to a domain not having enough values. As an example, consider the execution of the method *earn(0)* and the resulting path condition $\neg(\$0.points + \$3 > 100) \wedge \$0.points + \$3 >= 0$, shown in Fig. 7 (Trace I), with the initial domain values for variable $\$3$ including $[-1, 0, 1]$. If the branch $\neg(\$0.points + \$3 > 100)$ is flipped to cover a different execution path, $(\$0.points + \$3 > 100)$, and the SAT solver will find it to be trivially unsatisfiable since there is no domain value for *$3* that can make the condition evaluate to *true*. To prevent this, CCUJ analyzes the path condition and dynamically updates the domain based on the variable's relationship with other variables and concrete values in the path condition. In this example, CCUJ updates the domain for variable *$3* to include *[–1, 0, 1, 100, 101]*. A similar approach is used with non-primitive types, for example, for an association end *e1* with a multiplicity *0..**, CCUJ initially generates a domain with a predetermined number of values, e.g., *domain(e1) = [null, obj1, obj2]*. To update the number of objects in a non-primitive type domain, CCUJ requires the size property to be part of the path condition. For example, if the following path condition is observed, *$0.e1.size() < 4*, CCUJ updates the domain to include four or more objects. As a result, if the branch is flipped, an assignment is possible.

6.3 Symbolic Execution

To collect the symbolic path condition, CCUJ carries out the symbolic execution during the concolic execution of a method under test. While the symbolic execution of CCUJ is similar to the universal symbolic execution, CCUJ considers the method call during the symbolic execution since a *post-method* should be called after calling a method under test. More important, CCUJ implements the symbolic execution at the Java bytecode level since this approach was found to be more effective than symbolic execution at the Java source code level. We next illustrate the symbolic execution, which is done at line 6 and 9 in Fig. 11, as follows.

During the symbolic execution, CCUJ maintains a symbolic memory, denoted by *S*. S is implemented as a hash table mapping each program variable to a symbolic value. There are three basic operations that are performed on a symbolic memory. First, the creation of a new mapping relationship, *S = S +{[var->s]}*, adds a new mapping relationship *[var->s]* to *S*. Second the updating of a program variable with a new symbolic, *S = S[var->s]*, updates the only mapping relationship between *var* and *s* while preserving the remaining relationship in *S*. Third, the deletion of a program variable from a symbolic memory, *S = S-{[var]}*, removes the mapping relationship whose key is *var* from *S*. In each of these three operations, *var* is a program variable and *s* is a symbolic value. In order to obtain a symbolic value for a program variable such as a local variable and a field, CCUJ maintains its own symbolic stack frame including symbolic operand stacks. Since the Java Virtual Machine (JVM) is a stack machine CCUJ maintains a symbolic operand stack. As bytecode instructions are executed, symbolic values are added and removed from this stack in the same manner as the JVM manipulates concrete values. In this way, symbolic stack frames can mimic the stack frame used in JVM during runtime.

Next, we illustrate how a path condition is collected during concrete execution by hiding implementation details via a set of oracle functions. The function *getVar: Instructions → Variables* maps a bytecode instruction to a program variable and we write *inst.getVar()* to denote the variable related to instruction *inst*. This notation is also applied to the rest of oracle functions. The function *getCurOperandStack: Instructions → OperandStacks* maps the instruction to the current operand stack. The function *updateCurFrameStack: Instructions → OperandStacks* is used deal with method invocation and method return. When a method invocation instruction is applied, function *updateCurFrameStack* uploads a new stack frame for the callee method and makes it the current frame. The new stack frame also contains a new set of local variables and new operand stack. When a method return instruction is considered, the function *updateCurFrameStack* removes the current stack frame and returns to the caller's stack frame which is set to current.

```
1  symbolic_exe(inst)
2  { switch(inst) {
4    case load operations: //ILOAD, LLOAD, FLOAD, DLOAD, ALOAD etc.
5        if inst.getVar()  not exist S
6            S = S + {[getVar() -> si]}; i = i+ 1; //create a new mapping
7        else
8            si = S(getVar());
9        inst.getCurOperandStack().push(si);
10   case store operations:// ISTORE, LSTORE, FSTORE, DSTORE, ASTORE etc.
11       S= S[inst.getVar()->S.operandStack.pop()];//update the mapping
12   case branch operations: //IF_ICMPEQ, IF_ICMPNE, IF_ICMPLT etc.
13        p2=inst.getCurOperandStack().pop(); p1=inst.getCurOperandStack().pop();
14        path = collection_condition(inst); pathcondition.add(path);
15   case method call operations: //ARETURN,DRETURN,IRETURN, FRETURN etc.
16       inst.updateCurFrameStack();
17       for(each in inst.getParametersObjectRefs())
18           S=S+ {[each->each.getArgumentsObjecRefs()]} ; // create a new mapping
19           inst.getCurOperandStack().push(each.getArgument());
20   case return operations: //IRETURN, FRETURN, etc.
21       for(each_par in inst.getParameters())
22           S=S- {[each_par]}
23       value = inst.getCurOperandStack().pop(inst.getReturnValue())
24       inst.updateCurFrameStack();
25       inst.getCurOperandStack().push(value)
26   …..} }
```

Fig. 13. An outlined algorithm for CCUJ' symbolic execution.

Fig. 13 demonstrates a high-level overview of symbolic execution employed by CCUJ using several important JVM instructions. If an instruction is one of load-related instructions such as *iload*, CCUJ finds whether the variable related to this instruction exists in symbolic memory *S*. If not, CCUJ, at line 6 in Fig. 13, creates a new mapping, assigns a new symbolic value to this variable, and adds the mapping to *S*. Otherwise, CCUJ, at line 8 in Fig. 13, uses the variable as a key to find the symbolic value in symbolic memory *S*. Last, CCUJ pushes the symbolic value onto the operand stack, at line 9 in Fig. 13, to simulate the operand stack of JVM. If an instruction is one of store-related instructions such as *istore*, CCUJ simply pops the symbolic value from the operand stack and "updates" the symbolic value for the variable related to the current instruction as a key in symbolic memory *S*, at line 11 in Fig. 13. If the variable does not exist in *S*, the "update" action creates a new mapping relation and add it *S* in the same manner as load-related instructions. If an instruction is one of binary branch-related instructions such as *if_icmpeq*, CCUJ first pops the top two symbolic values from the operand stack, and calls method *collect_condition (inst)* to evaluate whether the top two values of the operand stack of JVM satisfy the current instruction, i.e. *inst*, or not, at line 13 of Fig. 13. Finally, using the evaluation result, CCUJ constructs the symbolic condition with the symbolic operand of the instruction and add the symbolic condition to the symbolic execution path, at line 14 of Fig. 13.

Another important kind of JVM instructions are method invocations and method terminations. When a method invocation instruction is met, CCUJ creates a new stack

```
if ( x * 2 > y * 2 )
```

a) Java source code

```
1 ILOAD 1        // load x
2 ICONST_2       // load constant 2
3 IMUL           // compute (x*2) as result1
4 ILOAD 2        // load y
5 ICONST_5       // load constant 5
6 IMUL           // compute (y*5) as result 2
7 IF_ICMPLE      // if condition is true, jump to
                    Else block
```

b) Java bytecode

Fig. 14. Java source code and bytecode.

frame which includes a new operand stack, at line 16 of Fig. 13, and makes is the current stack frame using function *updateCurFrameStack*. Similarly to JVM, CCUJ copies all symbolic values for the object references, i.e., *objectref*, local variables, and parameters on the new stack frame including the new operand stack, from line 17 to 19 of Fig. 13. When a method termination instruction is met, CCUJ removes all mappings related to the local variables and parameters of a called method from *S*, pops the return value from the current operand stack, updates the stack frames by setting the caller's stack frame current, and, finally, pushes the returned value onto the current operand stack, from line 23 to 25 of Fig. 13. To achieve the above functionalities for each JVM instructions, CCUJ instruments the original program binary with a set of instructions before or after each original JVM instruction at the preliminary testing preparation stage. The details of how to instrument instructions and the rest of JVM instructions regarding to the symbolic execution of CCUJ can be found in [17].

Instead of the symbolic execution at the Java source-level, CCUJ takes the advantages of the bytecode via the stack frame. For instance, evaluating a condition at the bytecode level is easier than the source code level. At the bytecode level, CCUJ only needs to consider the one or two concrete operands from the JVM operand stack to determine if the condition is true or false. Specifically, for condition *if* $(x^*2 > y^*5)$, CCUJ only retrieves the top values from the operand stack of JVM as they denote the value of x^*2 and y^*5 respectively (Fig. 14, line 7). Thus, the calculations of x^*2 and y^*5 are not necessary when the branch condition is evaluated. Each intermediate value has already been computed concretely by the JVM and symbolically by CCUJ as shown in lines 1-6 of Fig. 14. CCUJ simply evaluates whether the two operands satisfies ">" or not. Based on the concrete evaluation of the condition, CCUJ can get the symbolic values from its own symbolic operand stack and easily derives the symbolic path condition.

As another example, when a method call occurs in an expression, the symbolic execution at the bytecode level allows CCUJ to obtain the return value from an operand stack without introduction of a new local variable to hold a return value of a method.

6.4 Black Box Methods

Ideally, algorithm *Concolic_CCUJ* works well if we can execute a Java implementation in a full symbolic manner. However, when we applied CCUJ to the UML2 project, we found that there are some method calls to the APIs outside the UML metamodel such as Java collection APIs and UML EMF APIs which CCUJ does not instrument. There are several reasons that we did not instrument these APIs. First, they are not part of a design class diagram. The instrumentation of part of APIs such as UML EMF APIs does not solve the issue since the UML EMF implementation is also based on other Java APIs that cannot be instrumented due to restrictions such as licensing. Second, the instrumentation complicates the symbolic execution since more objects and their fields, such as ones used by the UML EMF to enhance the forward engineering features, must be tracked. As a result, we did not instrument any method call outside a design class diagram and left such methods as black box methods.

To symbolically execute a black box method, we employ the concrete execution to perform an approximate symbolic execution during the concolic execution. The approximate symbolic execution during the concolic execution allows CCUJ to update the symbolic memory using the concrete values. These concrete values are usually reference type values instead of primitive type values. Table 2 illustrates how CCUJ corrects the symbolic memory errors caused by the execution of a black box method. Assume the *BlackBoxMethodCall* method does some magic work which converts an array of ["a", "b", "c"] to ["b", "c", "c"]. However, during the concolic execution, CCUJ still can produce the correct symbolic memory. For instance, after calling the method *BlackBoxMethodCall,* there exists some inconsistency between the concrete memory and symbolic memory. However, when CCUJ executes a statement that reads the memory, such as a print statement, the inconsistency is identified. In this case, CCUJ employs the mapping relationship between a concrete variable and its symbolic value to update the symbolic memory to the value discovered. For example, when accessing the first element of array via *array [0]*, CCUJ finds the value in the concrete memory is the heap object storing "b" while the corresponding value in

TABLE 2
Memory Update during the Approximate Symbolic Execution

Code	Description	Concrete MemoryStructure	Symbolic MemoryStructure
public static void testBlackBoxWithHeap-Change() {String[] array = new String[] {"a", "b", "c"};	Array Initialization	["a", "b", "c"]	[A, B, C]
// [a, b, c] -> [b, c, c] BlackBoxMethodCall(array);	Black box method call	["b", "c", "c"]	[A, B, C]
array[2] = null	Direct change	["b", "c", null]	[A, B, Null]
System.out.println(array[0]);	Read value	["b", "c", null]	[B, B, Null]
System.out.println(array[1]);	Read value	["b", "c", null]	[B, C, Null]
System.out.println(array[2]); }	Read value	["b", "c", null]	[B, C, Null]

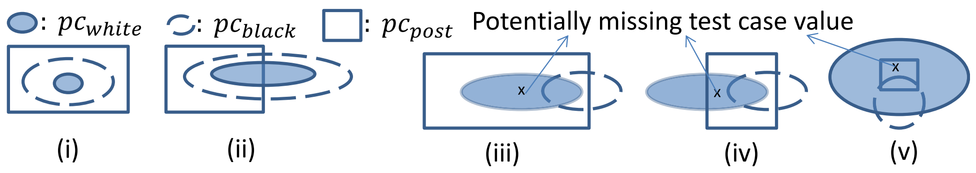


Fig. 15. Some scenarios causing true negative and false negative problems.

the symbolic memory is still A. Therefore, CCUJ updates the value to B in the symbolic memory.

6.5 Limitation

CCUJ employs the SAT solver Sat4j to find a new test case value when a path constraint is returned. One of the characteristics of Sat4j, like most SAT solvers, is that the satisfiability problem must be defined as a set of constraints over variables that can take one value from a finite domain [10]. This presents a problem when dealing variables that are bound to an infinite domain, such as floating point types. Due to this limitation, CCUJ cannot support float pointing types and non-linear constraints. However, when CCUJ checks the Java implementations derived from a class diagram, we find these limitations are not serious since most Java types CCUJ encounters are classes from the class diagram. While we cannot guarantee these limitations are not met by CCUJ, most applications using a forward engineering tool aim to find inconsistencies against the constraints related to a class in a class diagram.

While CCUJ adopts an approximate symbolic execution to deal with black box memory modifications, CCUJ still suffers the possibility of false negatives due to missing path conditions that occurred in the black box method. If CCUJ returns *false,* then a software fault is detected. In this case, we do not worry about the correctness of CCUJ since violation of a post-method is found. But, when CCUJ returns *true,* it might have missed some software faults in a Java program against its class diagram. The black box situation can be summarized as follows. Assume that, during the execution of one test case, all path conditions collected by a white-box are denoted as pc_{white}, and all path conditions missed by a black box are denoted as pc_{black}, and all path conditions for a post-method are denoted as pc_{post}. Note in CCUJ that the logical *implies* relationship between a path constraint for a method under test and a path constraint for a post-method plays an important role to find a new path constraint. Namely, the approximate symbolic execution in CCUJ attempts to use the relationship between pc_{white} and pc_{post} instead of pc_{white}, pc_{black}, and pc_{post} to find a new test case value that can follow a new path constraint. Consequently, CCUJ considers whether $pc_{white} \implies pc_{post}$, instead of $pc_{white} \wedge pc_{black} \implies pc_{post}$, is a tautology or not. In some cases when $pc_{white} \implies pc_{black}$ is a tautology, we find CCUJ works well when pc_{black} is missed from the consideration. For instance, if $pc_{white} \implies pc_{post}$ is a tautology, then $pc_{white} \wedge pc_{black} \implies pc_{post}$ must be tautology. Thus, the negation of pc_{white} is the same set of the negation of $pc_{white} \wedge pc_{black}$, as shown in Fig. 15i. Likewise, $pc_{white} \implies pc_{post}$ is not a tautology, CCUJ searches for a new test case that satisfies pc_{white} but not pc_{post}. Note that test cases that satisfy pc_{white} should satisfy $pc_{white} \wedge pc_{black}$, which would be considered if pc_{black} were available, as shown in Fig. 15ii. Consequently, CCUJ should not produce any false negatives if $pc_{white} \implies pc_{black}$ is a tautology.

However, CCUJ has a false negative problem when $pc_{white} \implies pc_{black}$ is not a tautology. For instance, if $pc_{white} \implies pc_{post}$ is a tautology, then, $pc_{white} \wedge pc_{black} \implies pc_{post}$ must be a tautology. In the approximate symbolic execution, CCUJ finds a test case value that satisfies $!pc_{white}$. But if path condition pc_{black} were available, a test case value that satisfies $!(pc_{white} \wedge pc_{black})$ would be sought. Therefore, the test case values that satisfy pc_{white} but not pc_{black} are missed for the consideration, as shown in Fig. 15iii. Likewise, if $pc_{white} \implies pc_{post}$ is not a tautology but $pc_{white} \wedge pc_{black} \implies pc_{post}$ can be a tautology, as shown in Fig. 15iv, then some test case values, which should satisfy $!(pc_{white} \wedge pc_{black})$, are missed when *CCUJ* tries to find a test case value that satisfies $pc_{white} \wedge !pc_{post}$. Similarly, some test case values can be missing in Fig. 15v. Thus, some software faults can be missed so the false negative problem might occur if $pc_{white} \implies pc_{black}$ is not a tautology.

7 Experiments

In this section, we present some experiments used to evaluate CCUJ from two aspects: effectiveness and efficiency. The project used for the evaluation is the UML2 implementation developed by the Eclipse Organization. There are several reasons we chose Eclipse's UML2 project. First, the UML metamodel is large and contains more than 300 metaclasses with numerous OCL constraints called well-formedness rules. Furthermore, Eclipse's UML2 implementation employs EMF to produce the Java skeletal program before all details are implemented. Second, as an open source project, Eclipse's UML2 project has an excellent bug report system that allows the developers and users to report and track each error in the project. In fact, CCUJ does detect various software faults in the UML2 project. The details of experiments such as OCL post-conditions, their Java post-methods and test case values with their path constraints can be found at [18].

7.1 Effectiveness

Next, we show the effectiveness of CCUJ through the detection of real software faults and through mutation testing.

7.1.1 Detection of Real Software Faults

In order to test whether CCUJ can find real software faults in Eclipse's UML2 projects, we initially targeted some UML abstractions that do not have the direct counterpart in Java. One example is the UML composition. Since Java does not have the corresponding syntax and semantics, composition must be enforced by the implementation. After further

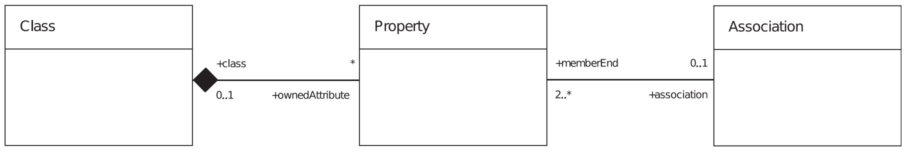


Fig. 16. Partial UML metamodel.

studying many existing approaches that claimed to recover UML composition by reverse engineering from a Java program, we found they do not strictly follow the semantics of UML composition [19], [20], [21]. The UML specification requires that "*... If a composite is deleted, all of its parts are normally deleted with it. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite...*" p. 41 [2]. However, many existing approaches require that all part objects cannot be accessed by any object except for its owner object, which is not supported by the UML specification. For instance, the class diagram in Fig. 16 excerpted from the UML specification shows that an object of class *Property*, which is owned by an object of class *Class*, can be accessed by an object of class *Association*. Therefore, when an owner object does not exist, all of its owned objects should not exist. Namely, all the links to the owned objects from other live objects should be removed. Assuming the method *destroy()* intends to implement the deletion of an owner object, Fig. 17 shows the composite property as a *post-condition* for the method *destroy()*. More importantly, the UML composition case is a representative example of checking a program developed from a Java skeletal program and mimics the testing process used in the MDE community. Specifically, Eclipse's EMF only generates the code for an association and programmers must add new implementation details to fulfill the constraints of UML composition. Thus, checking the constraints derived from UML composition is a good example to test the effectiveness of CCUJ.

After the above observation, we tried CCUJ on one of the UML2 projects, UML2 v1.1.1, and used as an example the metamodel fragment shown in Fig. 16. First, the OCL *post-condition* in Fig. 17 is automatically generated to test the composition property for the field *ownedAttribute* in metaclass *Class.* Based on the OCL *post-condition*, the equivalent Boolean Java method is generated, which is available at [18]. Next, CCUJ generates the initial minimal test case. In this example, it only includes an instance of metaclass *Class,* since the multiplicity of the association between *Class* and *Property* is * (0 to many), making it optional. After that, CCUJ calls the method under test, *destroy()* and the generated Java Boolean *post-method*, *post_destroy()* to test whether *destroy()* satisfies the *post-condition*.

CCUJ did detect that method *destroy()* does not satisfy the *post-condition*, indicating an implementation fault for UML composition in UML2 v1.1.1. The fault was confirmed by one of UML2 project members. The cause of the implementation error is that the *destroy()* method iteratively checks

```
Context Class::destroy():Boolean
post: seslf.ownedAttribute@pre->forAll(p |
      p.association@pre.memberEnd->excludes(p))
```

Fig. 17. OCL post-condition for UML composition.

each object contained in the resource, which is supposed to contain all the instantiated objects, and remove their links to the owned objects being destroyed. The resource object, as part of EMF metamodel, however, did not automatically store all instantiated owned objects in the resource object appropriately. For this OCL constraint, CCUJ totally finds 246 error instances. CCUJ was applied to test the composition in a later version UML2 v2.2.1 and the previous fault has been fixed.

Next, we also applied CCUJ to the UML2 project v4.0.2 to check 55 well-formedness rules in the UML specification [2]. Among these well-formedness rules, CCUJ detected a software fault on the OCL *post-condition* when implementing method *isAttribute()* in class *Property*. The well-formedness rule of class *Property* is defined as an OCL *post-condition* (p. 125 [2]) for the method. After translating the OCL *post-condition* into a Java Boolean *post-method*, CCUJ is called on the method under test, *isAttribute()* and its *post-method*, *post_isAttribute()* and a fault is detected. The fault lies in the fact that the implementation only checks the *non-navigable* inverse references to property *p*. That is, the references in which an object (*obj1*) can access *p*, but *p* cannot directly access the object *obj1.* Since the reference *attribute* in class *Classifier* is a *navigable* inverse reference, it was ignored, and the method failed to return *true* when *c.attribute->includes(p)* is *true*. The problem was confirmed and fixed[3] by the developers.

Last, we also applied CCUJ to check some validation methods, each of which is used to check a well-formedness rule of a metaclass according to the UML specification. Thus, some validation methods are regarded as a *post-method* while the others are regarded as an invariant, depending on how the corresponding well-formedness rule is defined in the UML specification. When applying CCUJ to check the implementation of some validation methods, we found that some well-formedness rules in the UML specification are only described in the natural language without the corresponding OCL constraints. In these cases, we gave the OCL constraints first. Otherwise, the OCL constraints in the UML specification are used. Next, we converted these OCL constraints to a *post-condition* of their corresponding validation methods by using the keyword "*result*" in OCL. More specifically, the Boolean value returned by a validation method should conform to the corresponding well-formedness rule in the UML specification.

To illustrate to the process for checking a validation method, consider an invariant for metaclass Connector. The UML specification states, "*A delegation connector must only be defined between used Interfaces or Ports of the same kind (e.g., between two provided Ports or between two required Ports)*" (p. 125 [2]). Accordingly, we thus gave the corresponding OCL constraint. The UML2 project has an invariant validation

3. https://bugs.eclipse.org/bugs/show_bug.cgi?id=407028 [accessed 7-June-2013]

method *validateBetweenInterfacesPorts()* in metaclass *Connector* that enforces the above invariant. To test *validateBetweenInterfacesPorts()*, we derived a *post-condition* for method *validateBetweenInterfacesPorts()*, using the keyword *"result"* to ensure that return value conforms to the original invariant in the UML specification. CCUJ found a software fault in UML2 v4.1.0 related to the *validateBetweenInterfacesPorts()* but the fault was fixed in UML2 v4.2.0. Due to space, readers are referred to [18] for details.

Actually, all the above software faults cannot be effectively revealed by other current approaches. As aforementioned, the testing-based approaches including the symbolic testing technique suffer either runtime exceptions or false positives. Likewise, the approaches using the small scope hypothesis such as glass box testing approach [12] and Korat [22] are also hampered by the auxiliary fields generated by a forward engineering tool. Again, these auxiliary fields cannot be set to any value in a field domain used by a SAT solver, since their specific values are set based on the structure of a class diagram. Furthermore, the Java pre-method translated from an OCL pre-condition is not able to detect an illegal value for an auxiliary field because it occurs in the neither OCL pre-condition nor the Java pre-method. Runtime exceptions as well as false positives prevent these approaches from revealing software faults in an effective fashion.

7.2.2 Mutation Testing

Mutation testing is originally employed to evaluate the quality of software testing. The main idea of mutation testing is to introduce some modifications that mimic programmer's error in a program under test. Each modified version of a program under test is called a mutant. A test *t* is said to kill a mutant if the output of *t* on the original program is different from the output of *t* on the mutant. A mutation testing process is used to decide whether a test process should continue by evaluating the ratio of killed mutants to the total number of mutants against a defined threshold. However, we employed mutation testing to evaluate the effectiveness of CCUJ to detect the errors compared to some other approaches that use the small scope hypothesis.

To carry out mutation testing, we chose six types of mutation operations, which include 1) Absolute Value Insertion, 2) Arithmetic Operator Replacement, 3) Relational Operator Replacement, 4) Conditional Operator Replacement, 5) Assignment Operator Replacement, and 6) Unary Operator Insertion. In order to remove any bias, we considered all 21 OCL constraints given in chapter 2 of [3]. Applying the six types of mutation operations to the above 21 methods, we totally produced 383 mutants. We applied CCUJ to test these 383 mutants and CCUJ can kill 360 mutants and the percentage of mutant killed is 93.8 percent. All missing mutants where equivalence to the original program, and, therefore, did not represent a bug. For instance, *index* $! = -1$ is equivalent to *index* > -1 when replacing $>$ with $! =$, since, in method *Property.getOpposite()*, variable *index* cannot be less than –1. On the other hand, when we employed the glass box testing approach as a representative one based on the small scope hypothesis with a scope of 6 for each variable only 242 mutants (73.8 percent) were killed. The details of the mutation testing including all other non-killed mutants can be found in the Appendix, available in the online supplemental material, as well as online [18].

7.2 Efficiency

The efficiency of CCUJ was investigated by comparing the number of test cases generated with that of the two aforementioned approaches, which are a glass box testing approach and Korat. One reason for this selection is that these two approaches employ different methods to generate test case values. We note that the number of test cases determines the number of times that the method under test must be executed. In general, the smaller the number of test cases, the greater the efficiency. Both of Korat and the glass box testing approach use finitization to limit the number of values that can be assigned to a field. However, in the glass box testing approach, the search space is pruned by ignoring fields not accessed during the execution of the method under test. While the small scope hypothesis is the assumption of both approaches, they cannot be as effective as CCUJ since CCUJ explores different test cases to achieve all possible execution paths. To show the percentage of the execution paths which can be covered by the small scope hypothesis, we applied the six types of mutation operations to the aforementioned 21 OCL constraints. Consequently, we are able to check all 383 mutant methods and we find that the average percentage of the execution paths covered by the small scope hypothesis based approach, using 6 as a scope value, is only 70 percent while CCUJ explores all execution paths. The details of the experiment on the branch coverage can be found in the Appendix, available in the online supplemental material, as well as online [18].

To compare CCUJ with Korat on the generation of test input state space, we consider the multiplicity and navigability constraints in the class diagram. These constraints are converted into an invariant method *repOK()* that Korat uses to generate all test cases. For example, in Fig. 16, *repOK ()* in *Class* checks the multiplicity and navigability on *Property*'s end. Method *repOK()* in *Property* checks the multiplicity and navigability on both *Class'* and *Association*'s end.

The results of the comparison for nine methods from the UML2 project are shown in Table 3. The second column in the table shows the number of classes involved on each of the tests, which represents a fragment of the UML metamodel. Considering all the metamodel classes would greatly increase the number of test cases generated by the Korat approach, since it would recursively call *repOK()* in all the classes involved. The third column displays the finitization values used for both Korat and glass box testing approaches. The finitization value is the number of different values used for each field during the test case generation. Our approach, shown in column six, however, considers all possible values. The fourth column shows the number of test cases generated by the glass box testing approach. Depending on the finitization and the number of fields that are touched during the execution, the number of generated test cases can vary for the glass box testing approach. For example, when the finitization for method *hasVisibilityOf()* is 3, the number of test cases generated by the glass box testing approach is 27. The fifth column shows the number of test cases generated by the

TABLE 3
Test Case Generation Comparison

Test	No. Classes	Number of Test Cases					Time (ms)		
		Finitization # of possible values per field	Glass Box	Korat	Finitization # of possible values not bounded	Our Approach	Glass Box	Korat	Our Approach
AssociationClass::hasVisibilityOf()	2	3	27	6,561	∞	2	864	209,952	67
	2	4	64	65,536	∞	2	2,048	2,097,152	67
	2	5	125	390,625	∞	2	4,000	12,500,000	67
AssociationClass::qualifiedName()	2	3	27	6,561	∞	2	459	111,537	80
	2	4	64	65,536	∞	2	1,088	111,412	80
	2	5	125	390,625	∞	2	2,125	6,640,625	80
Package::importedMember()	2	3	9	81	∞	3	486	4,374	128
	2	4	16	256	∞	3	864	13,824	128
	2	5	25	625	∞	3	1,350	33,750	128
Property::deployedElement()	2	3	9	27	∞	3	1,315	945	250
	2	4	16	64	∞	3	560	2,240	250
	2	5	25	125	∞	3	875	4,375	250
ProtocolStateMachine::conformsTo()	4	3	81	59,049	∞	2	2,106	1,535,274	63
	4	4	256	1,048,576	∞	2	6,656	time out	63
	4	5	625	9,765,625	∞	2	16,250	time out	63
Classifier::isTemplate()	4	3	108	2,916	∞	6	2,052	55,404	114
	4	4	256	16,384	∞	6	4,864	311,296	114
	4	5	500	62,500	∞	6	9,500	1,187,500	114
StateMachine::ancestor()	8	3	27	6,561	∞	10	621	150,903	230
	8	4	64	65,536	∞	10	1,472	1,507,328	230
	8	5	125	390,625	∞	10	2,875	8,984,375	230
Classifier::maySpecializeType()	4	3	3	6,561	∞	3	36	78,732	63
	4	4	4	65,536	∞	3	48	786,432	63
	4	5	5	390,625	∞	3	60	time out	63
Element::destroy()	3	3	27	531,441	∞	3	648	time out	240
	3	4	64	16,777,216	∞	3	1,536	time out	240
	3	5	125	244,140,625	∞	3	3,000	time out	240

Korat approach. The number of test cases is determined by number of parameters, number of classes, and the associations between them, where each of the fields can take up to 3-5 values depending on the finitization. This results in a large number of test cases since the Korat approach tries all possible combinations for all the fields involved. However, while CCUJ does not consider any finitization number, it produces a much smaller number of test cases, as shown in column seven. Finally, the last three columns show the time to generate these test cases by the glass box testing approach, the Korat approach, and CCUJ. From this information, we are confident about the efficiency of CCUJ since it employs a more efficient pruning technique to reduce test case input space. The only exception is method *maySpecializedType()* in class *Classifier*. In this case, the glass box testing approach generates a similar number of test cases, and without the overhead of the path condition analysis, it does so in a smaller amount of time. The reason of this is that there is only one field read during the execution of the method and the method is simple enough that all branches are covered while trying all the finitization values. However, this is not a common scenario. Typically, methods are more complex and more fields are read during an execution. More important, CCUJ considers all possible values in one execution and the correctness of the method is guaranteed with the smallest possible number of test cases.

8 Related Work

Consistency checking among different UML diagrams has been a hot topic since UML became the de facto industry standard for object-oriented software design. There is a great amount of related literature. UML class, sequence, and state machine diagrams have been popular subjects for the investigation of inconsistency checking by many researchers. All these approaches can fall into two categories, vertical consistency checking and horizontal consistency checking. Horizontal inconsistencies exist in two or more UML diagrams showing the different aspects of a software system at a time. Vertical inconsistencies are introduced by different versions of a UML model, each of which represents a different phase during the software system's evolution. Most approaches map different UML diagrams to some intermediate representations such as a graph structure or a formal language to detect inconsistencies while some others do not use any intermediate representation.

For instance, Ehrig and Tsiolakis check the consistency between class and sequence diagrams of a software system using a common graph structure [23]. Rasch and Wehrheim investigate the UML class and state machine diagrams to find inconsistencies after both diagrams have been translated to CSP-OZ [24]. Li et al. search for inconsistencies among the UML class, use cases, and activity diagrams via a formal language called the Object Oriented Specification Language (OOL) [25]. In principle, these approaches target the horizontal inconsistencies in UML diagrams showing different aspect of a software system.

Some researchers employ some consistency rules that relate different elements in various diagrams to detect inconsistencies. Usually, these approaches emphasize the efficiency of consistency checking. For example, Egyd uses OCL constraints to show the consistency rules that a system should satisfy and investigates how a change affects the previously evaluated rules [28]. Girschick uses UML class diagrams to detect inconsistencies between two versions by means of coloring different class properties [29]. Nentwich et al. proposes a novel approach to check inconsistencies in software artifacts including UML diagrams based on a XML-based environment using consistency rules expressed in a uniform manner [30]. Most of these approaches can detect at least vertical inconsistency if not both vertical and horizontal. All of these approaches study the static information in UML diagrams rather than the dynamic behavior of a software system and, thus, are limited in checking the consistency of a Java implementation against its class diagram.

There exist some vertical consistency checking approaches that do consider a Java implementation. For instance, Pires et al. investigate the test case generation for a Java implementation that is derived from a class diagram [26]. The difference between their approach and CCUJ is that Pires et al. provide a framework to test whether each attribute and method in a Java program conforms to their counterpart in a class diagram. For instance, the framework can determine whether an attribute in a Java program has the same declaration type as the attribute's counterpart in a class diagram. We observed that attributes and methods in a class diagram can be automatically converted by many forward engineering tools to Java class elements in a skeletal program. The more challenging issue is to ensure that manual implementation added afterward conforms to the class diagram and especially to class constraints. Ciraci et al. present another approach testing a Java implementation with a concentration on the conformance checking between a combined class diagram and sequence diagram as a design model and its Java implementation [27]. More specifically, Ciraci et al. use sequence diagrams to generate an execution tree which is employed to check any recorded execution of a Java implementation that violates a sequence diagram. In their case, the order of method calls is treated as a constraint that should be enforced. But they do not consider more general constraints such as OCL constraints given as part of the design model.

Recovery of some specific UML properties from a Java implementation such as association related properties has been also studied by some researchers. In addition to [21], Milanova studies how to reverse Java software to UML composition via static ownership inference based on points-to analysis [28]. Kollmann and Gogolla present definitions and identification algorithms for implementation-level association, aggregation, and composition relationships via dynamic analysis [29]. All of these approaches emphasize the reverse engineering of an implementation and cannot be applied to check whether a Java implementation satisfies all properties defined in its design class diagram. Actually, these approaches can be regarded as a special feature of CCUJ when the UML properties can be converted to OCL constraints.

Another related work includes UML model-based software testing techniques that have become an effective approach. There is a plethora of testing techniques and criteria based on UML diagrams. Most UML diagrams considered by researchers include use case and interaction diagrams. Abdurazik et al. present a test criteria based on UML collaboration diagram allowing a test case to be generated from a design model instead of code or specification [30]. Briand et al. propose a novel approach to enhance interaction testing by combining state-based behavior and data flow information [31]. More specifically, they use state machine and sequence diagrams to produce a control-flow graph that can further generate integration test cases. However, most UML-based based software testing approaches just concentrate on the test case generation from UML diagrams and ignore the implementation details. Thus, these approaches cannot be applied to perform consistency checking between a Java implementation and its design class diagram.

Turner et al. present a new type of diagram called Visual Constraint Diagrams (VCD), which extends UML class and object diagrams [32]. VCD allows developers to not only document object models visually but also express constraints over the instantiation of the models. They claim that the constraints in VCD can be used to assess the runtime behavior of a program mainly as the role of a debugger. CCUJ, however, actually is not a debugger but a model checker that ensures the conformance relationship between the dynamic behavior of a Java program and its design class diagram.

In the object-oriented programming language community, we notice that many researchers have attempted to raise the abstraction level of object-oriented programming languages such as Java so that the semantics of these abstractions is close to UML semantics. For example, the object ownership model [20], which is close to UML composition properties, has been widely studied. Basically, there are two main approaches to object ownership in the literature: enforcing some program conventions within an existing syntax of a programming language, or significantly revising the syntax of a programming language to allow ownership support. The first approach usually requires programmers to follow a set of specific conventions so that object ownership can be checked [33], [34]. The second approach usually adds ownership to the syntax of a language and ownership is expressed explicitly within the type systems of the language [20], [35]. While the introduction of a new structure or syntax in Java can reduce the workload to check the inconsistency, CCUJ aims at the consistency checking based on the current syntax and semantics of Java and, thus, can be applied to the existing software systems.

Some researchers attempt to check the correctness of UML/OCL models, which inspire the development of CCUJ. Gogolla et al. present a UML-based Specification [26] Environment (USE) in which users can simulate the behavior of UML models and check whether each state represented as an object diagram satisfies OCL invariants, *preconditions*, and *post-conditions* during the simulation [36]. Since USE targets the correctness of a design model, it does not consider whether an implementation faithfully satisfies a design model or not. Furthermore, although each heap configuration in CCUJ can be regarded as an object diagram, USE does not discuss the use of different testing criteria to cover different execution paths during simulation. Yu et al. present an approach to check size property of collection types in OCL using the invariants of a class and the *pre-* and *post-conditions* of all methods in that class [9]. While this approach can be applied to an implementation, the authors just concentrate on the inconsistencies in the size properties of OCL constraints in the form of class invariants, method *pre-conditions*, and *pre-conditions*, and method *post-conditions*, and ignore whether OCL constraints are correctly implemented by an implementation.

Testing model transformation is another area related to our work. Testing model transformation seeks to reveal a fault in a transformation of a model. If a transformed model introduces a fault, it can propagate to other models during the later phases of a software development process. As a result, it is hard to detect and fix the fault. To find such a fault during the transformation process, many researchers investigate various approaches to test whether a transformation can produce a correct target model or not. Like CCUJ, testing model transformation faces the infinite input domain where a transformation is tested. Benoit discusses some criteria for metamodel coverage to select test models for the transformation [37]. Gonzalez and Cabot propose an approach for testing ATL transformations [38]. They consider the application of constraint solvers to automatically generate test source models. Schonbock et al. propose $\text{TETRA}_{\text{Box}}$ as a generic framework for execution-based white box testing of transformation languages [39]. In addition to providing the automatic generation of test models, $\text{TETRA}_{\text{Box}}$ uses a Pattern-based Modeling Language for Model Transformations to provide a trace failure if a test model fails. All these approaches target a transformation program, which is Φ in this paper, while CCUJ studies a Java program that is developed on a skeletal code generated by a transformation program Φ.

Software testing is also another area related to our work. While inspired by the latest development in software testing techniques, none of these methods can be directly applied to consistency checking between a Java implementation and its design class diagram. For example, Java Pathfinder is an important testing tool that generates as many test cases as possible to uncover some erroneous runtime behavior such as a runtime exception and the failure of an assert statement [40]. The Java program investigated by CCUJ, however, does not have specific statements to reach; instead CCUJ is interested in the consistency between a Java implementation and its design class diagram in terms of constraints given in a class diagram. Darga and Boyapati present an efficient way to check data structure properties [12]. While their goal of checking data structure properties is different from CCUJ, both approaches attempt to find an efficient method to generate as few test cases as possible without loss of any testing ability. But their approach, based on the *don't care* fields to prune the test cases, cannot be as efficient as CCUJ does since CCUJ considers both the symbolic execution path from the method under test and its *post-method*. Actually, the approach employed by CCUJ to study the implication relationship between the two symbolic execution path can eliminate all those test cases pruned by their approach based on the *don't care* fields. Marinov et al. present a tool called TestEra which uses the Alloy language to specify the constraints that state the relationship between inputs and outputs of a program under test [41]. TestEra generates all non-isomorphic inputs within a given bound on the input size, runs each input on the program under test, and checks whether each pair of the inputs and outputs satisfies the constraints. However, CCUJ does not adopt the bounded-exhaustive testing requirement so there is no restriction on the input space. Thus, more execution paths of a method under test can be explored. Furthermore, CCUJ considers the relationship between the executions of a method under test and its *post-method* to prune the test cases which are guaranteed to have the same behavior.

Conformance evaluation has been widely employed by researchers to attempt to find errors in a software model. The Meta-Object Facility (MOF), proposed by the Object Management Group (OMG), is designed as four-layered architecture. The most prominent example of an M_2 layer MOF model is the UML metamodel which describes UML itself. Since an M_i layer model describes the elements of M_{i-1} layer model, the conformance evaluation checks whether all elements of M_{i-1} layer model satisfy the corresponding M_i layer model. Richters et al. use the relationship between a class diagram, i.e., a M_1 layer model, and an instance diagram, i.e., an M_0 layer model to ensure whether a scenario produced by the behavior of a UML model satisfies the class diagram. Also, France et al. propose a new metamodel called the Role-Based Metamodeling Language (RBML), which is an M_2 layer model, to specify software design patterns [42]. Based on the RBML, Kim et al. evaluate a class model as a M_1 layer model via conformance evaluation to find any errors caused by some elements in the class model that violates the metamodel [43]. Compared to this approach, CCUJ can be regarded as the conformance checking that seeks to efficiently generate all M_0 layer models-heap configurations—by considering the dynamic behavior of a Java program and check them against the class diagram.

9 Conclusions and Future Work

With the rapid development of MDE, many software systems are implemented based on skeletal code generated by some forward engineering tools. Under the assumption of the correct behaviors of object creation and association link creation/modification in the generated skeletal code, we propose a model-based approach to check the consistency between a UML class diagram and its final Java implementation. More specifically, CCUJ ensures that each *post-condition* of an operation in a UML class diagram is satisfied by the corresponding Java implementation. More importantly, CCUJ can efficiently prune a large number of test cases using the symbolic execution of a method under test and its post-method and the relationship between these two methods' path constraints. CCUJ has been successfully applied to some industry-size projects such as the UML2 projects. Some real software faults have been detected by CCUJ and confirmed by the developers. The empirical data shows that CCUJ is more efficient than many existing testing tools in the generation of test cases. We are confident that CCUJ can be widely used to test a software system that is developed under the MDE environment.

In the future, we plan to apply CCUJ to test a generated skeletal framework produced by forward engineering tools.

As part of testing model transformation, testing a generated skeletal code is also extremely important since the generated code is the initial mainframe for a final implementation provided by programmers. To this end, the methods implementing object creation and association link creation and modification through the use of the constructors and setter methods are the ones CCUJ will test. This will complement what CCUJ has done in this paper.

In Memory. We are saddened by the passing of our dear colleague and coauthor, Professor Robert B. France. Professor France worked with great perseverance and optimism on this project in the past few years. His deep insight and passion regarding the application of MDE in software development shown in this project will continue to guide and inspire us for years to come.

ACKNOWLEDGMENTS

The authors thank many anonymous reviewers for their constructive and detailed comments during the long journey of this project. Additionally, they really appreciate their colleagues Drs. Betty Cheng, Laura Dillon, Tao Xie and Zijiang Yang for their feedback, discussion, and support of this project. Dr. Li is supported by the National Natural Science Foundation of China (No. 61472406) and the Open Project of Shanghai Key Lab. of Trustworthy Computing (No.07dz22304201302).

REFERENCES

[1] Reactive Systems Inc. (2003). Software Testing and Validation with Reactis [Online]. Available: http://www.reactive-systems.com/

[2] OMG. (2011). OMG Unified Modeling Language (OMG UML), Superstructure Specification (Version 2.4.1)

[3] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2 ed. Boston, MA, USA: Addison-Wesley, 2003.

[4] D. Larsson and W. Mostowski, "Specifying JAVACARD API in OCL," *Electronic Notes Theoretical Comput. Sci.*, vol. 102, pp. 3–19, 2004.

[5] IBM, *Rational Software Architect*, 2010.

[6] Y. Kannan and K. Sen, "Universal symbolic execution and its application to likely data structure invariant generation," in *Proc. Int. Symp. Softw. Testing Anal.*, 2008, pp. 283–294.

[7] H. Chavez, W. Shen, R. France, and B. A. Mechling, "An approach to testing Java implementation against its Unified Modeling Language," in *Proc. ACM/IEEE Int. Conf. Model Driven Eng. Languages Syst.*, Miami, FL, USA, Oct. 2013, pp. 220–236.

[8] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge Univ. Press, 2008.

[9] F. Yu, T. Bultan and E. Peterson, "Automated size analysis for OCL," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2007, pp. 331–340.

[10] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2.," *J. Satisfiability, Boolean Model. Comput.*, vol. 7, pp. 59–64, 2010.

[11] A. Igarshi, B. Pierce and P. Wadler, "Featherweight Java: A mimimal core calculus for Java and GJ," *ASM Trans. Program. Languages Syst.*, vol. 23, no. 3, pp. 396–450, 2001.

[12] P. T. Darga and C. Boyapati, "Efficient software model checking of data structure properties," presented at the *21st ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, 2006.

[13] K. Sen, D. Marinov and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2005, pp. 263–272.

[14] S. Anand, M. Naik, H. Yang, and M. Harrold, "Automated concolic testing of smartphone apps," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2012, pp. 1– 11.

[15] The Eclipse Foundation. (2013). Eclipse Modeling - MDT [Online]. Available: http://www.eclipse.org/modeling/mdt/ [Accessed 2013].

[16] OMG, "Object Constraint Language, version 2.3.1," 2006.

[17] B. A. Mechling, "Concolic execution of Java bytecode," Dept. Comput. Sci., Western Michigan Univ., Tech. Rep. WMU-CS TR/15-01-01, Kalamazoo, MI, USA, 2015.

[18] CCUJ Project. (2014, Oct.). [Online]. Available: http://cs.wmich.edu/~h6chavezchav/symbolic/

[19] F. Barbier, B. Henderson-Sellers, A. Le Parc-Lacayrelle, and J.-M. Bruel, "Formalization of the whole-part relationship in the Unified Modeling Language," *IEEE Trans. Softw. Eng.*, vol. 29, no. 5, pp. 459–470, May 2003.

[20] C. Boyapati, B. Liskov, and L. Shrira, "Ownership types for object encapsulation," in *Proc. ACM Symp. Principles Program. Languages*, New York, NY, USA, 2003, pp. 213–223.

[21] Y. Gueheneuc and H. Albin-Amiot, "Recovering binary class relationships: Putting icing on the UML cake," in *Proc. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Languages, Appl.*, New York, NY, USA, 2004.

[22] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, New York, NY, USA, 2002, pp. 123–133.

[23] H. Ehrig and A. Tsiolakis, "Consistency analysis of UML class and sequence diagrams using attributed graph grammars," in *Proc. Workshop Graph Transformation Syst.*, 2000, pp. 77–86.

[24] H. Rasch and H. Wehrheim, "Checking consistency in UML diagrams: Classes and state machines," in *Proc. 6th IFIP WG 6.1 Int. Conf. Formal Methods Open Object-Based Distrib. Syst.*, 2003, pp. 229–243.

[25] X. Li, Z. Liu, and J. He, "Consistency checking of UML requirements," in *Proc. 10th Int. Conf. Eng. Complex Comput. Syst.*, Shanghai, China, 2005, pp. 411–420.

[26] W. Pires, J. Brunet, and F. Ramalho, "UML-based design test generation," in *Proc. ACM Symp. Appl. Comput.*, 2008, pp. 735–740.

[27] S. Ciraci, S. Malakuti, S. Katz, and M. Aksit, "Checking the correspondence between UML models and implementation," in *Proc. 1st Int. Conf. Runtime Verification*, 2010, pp. 198–213.

[28] A. Milanova, "Precise identification of composition relationships for UML class diagrams," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, New York, NY, USA, 2005, pp. 76–85.

[29] R. Kollmann and M. Gogolla, "Application of UML associations and their adornments in design recovery," in *Proc. 8th Working Conf. Reverse Eng.*, 2001, pp. 81–90.

[30] A. Abdurazik and J. Offutt, "Using UML collaboration diagrams for static checking and test generation," in *Proc. 3rd Int. Conf. Unified Model. Language*, 2000, pp. 383–395.

[31] L. Briand, Y. Labiche, and Y. Liu, "Combining UML sequence and state machine diagrams," in *Proc. 8th Eur. Conf. Model. Found. Appl.*, 2012, pp. 74–89.

[32] C. J. Turner, T. N. Graham, H. D. Stewart, and A. G. Ryman, "Visual constraint diagrams: Runtime conformance checking of UML object models versus implementations," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2003, pp. 271–276.

[33] D. Clarke, M. Richmond, and J. Noble, "Saving the world from bad beans: Development-time confinement checking," in *Proc. ACM Conf. Object-Oriented Program., Syst., Languages, Appl.*, Oct. 2003, pp. 374–387.

[34] J. Hogg, "Islands: Aliasing protection in object-oriented languages," in *Proc. ACM Conf. Object-Oriented Program., Syst., Languages, Appl.*, Nov. 1991, pp. 271–285.

[35] J. Aldrich, V. Kostadinov, and C. Chambers, "Alias annotations for program understanding," in *Proc. ACM Conf. Object-Oriented Program., Syst., Languages, Appl.*, 2002, pp. 311–330.

[36] M. Gogolla, J. Bohling, and M. Richters, "Validation of UML and OCL models by automatic snapshot generation," in *Proc. 6th Int. Conf. Unified Model. Language*, 2003, pp. 386–398.

[37] B. Benoit, "Testing model transformations: A case for test generation from input domain models," in *Model Driven Engineering for Distributed Real-Time Embedded Systems*. Paris, France: Hermes, 2009.

[38] C. A. Gonzalez and J. Cabot, "A TL test: A white-box test generation approach for ATL transformations," in *Proc. ACM/IEEE Int. Conf. Model Driven Eng. Languages Syst.*, 2012, pp. 449–464.

[39] J. Schonbock, G. Kappel, M. Wimmer, A. Kusel, and W. Retschitzegger, "TETRABox – A generic white-box testing framework for model transformations," in *Proc. Asia-Pacific Softw. Eng. Conf.*, 2013, pp. 75–82.

[40] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Softw. Eng. J.*, vol. 10, no. 2, pp. 203–232, 2003.
[41] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2001, pp. 22–31.
[42] R. France, D. Kim, S. Ghosh, and E. Song, "A UML-based pattern specification technique," *IEEE Trans. Softw. Eng.*, vol. 30, no. 3, pp. 193–206, Mar. 2004.
[43] D. Kim and W. Shen, "Evaluating pattern conformance of UML models: A divide-and-conquer approach and case studies," *Softw. Quality J.*, vol. 16, no. 3, pp. 329–359, 2008.

Hector M. Chavez received the BCS degree in computer science from the Instituto Tecnologico de Morelia, Mexico, and the MS and PhD degrees in computer science from Western Michigan University. His research interests include automated software testing and verification, model-driven development, and model transformation techniques. He currently holds a position with the Global Software Controls Group at Dematic Corp.

Wuwei Shen received the PhD degree in computer science from the Department of Electrical Engineering and Computer Science, University of Michigan in 2001. He is currently an associate professor at Western Michigan University. His research interests include object-oriented analysis and design modeling, model consistency checking, model-based software testing, assurance-based software development, and software certification. He is a member of the IEEE.

Robert B. France received the BS degree in 1984 from the University of the West Indies, Trinidad, and the PhD degree in 1990 from Massey University, New Zealand. He passed away on February 15, 2015 and before he passed away, he was a professor at the Department of Computer Science, Colorado State University. His research on model-driven software development focused on providing software developers with mathematically based software modeling languages and supporting analysis tools that they can use to specify and analyze critical software properties. He was the founding editor-in-chief of the Springer journal *Software and Systems Modeling*, and a founding steering committee member of the international conference series on Model Driven Engineering Languages and Systems (MODELS). In 2008, he and his co-authors (Andy Evans, Kevin Lano, Bernard Runpe) received the MODELS 2008 Ten Year Most Influential Paper Award for the paper "The UML as a Formal Modeling Notation". In 2013, he received a five-year International Chair at INRIA, the French National Institute for research in Computational Sciences. He received a 2014 Senior Dahl-Nygaard Award for his research contributions by AITO, the group that organizes the European Conference on Object-Oriented Programming. He also received a Colorado State University, College of Natural Sciences Professor Laureate in 2014. In addition, the Institute of Caribbean Studies honored him with the prestigious Excellence in Science and Technology Award for 2014.

Benjamin Mechling received the bachelor's and master's degrees in computer science from Western Michigan University. He worked on an approach to concolic execution of Java bytecode. He is currently working on database programming for a web application.

Guangyuan Li received the PhD degree in computer science from the Institute of Software, Chinese Academy of Sciences (CAS) in 2001. He is currently an associate professor in the State Key Laboratory of Computer Science, Institute of Software, CAS. His research interests include formal methods, temporal logics, and model checking, in particular, the formal verification tools for real-time systems.