Runtime Checking of UML Association-Related Constraints

Kun Wang and Wuwei Shen
Department of Computer Science
Western Michigan University
Kalamazoo, Michigan 49008, USA
{kun.wang, wuwei.shen}@wmich.edu

Abstract

UML class models are important design artifacts used as blueprints of software systems to be built. Yet, implementations are often inconsistent with their models. Although many techniques have been proposed to tackle this problem, some dynamic aspects of a class model are still hard to be verified using existing techniques. In this paper, we present an approach to checking the UML association-related constraints during a program's execution. Our approach instruments event notification mechanism into Java bytecode and verifies the constraints imposed by design-level associations (including compositions), when certain events occur. The empirical studies show that our approach can help efficiently detect inconsistencies between a UML class model and its implementation.

1. Introduction

UML class models are important design artifacts during software development. Yet, implementations are often inconsistent with their models. These inconsistencies are hard to be detected by software engineers as a system becomes complex. As a result, the runtime behavior of a software system is not guaranteed to be consistent with its model.

The scientific literature is rich with techniques and tools, most of which use static approaches to reverse engineering class models from the code [19, 4, 6, 10, 13, 9]. However, a class model contains some dynamic constraints whose implementation is hard to be statically verified. For instance, a composition requires that a part have coincident lifetime with its owner. Using static techniques to verify such dynamic constraints (e.g., object lifetime) would be imprecise and expensive, if not impossible.

For example, Milanova [13] proposed a static ownership inference technique to identify composition relationships. In her work, a part cannot be exposed outside of its owner, i.e., a part may be accessed only by its owner as well as other objects within its owner's boundary. However, as long as an owner is *alive*, the composition's definition does not prevent a part from being accessed by any other objects through non-composition *links* [16]. In the UML 2.0 specification [16], for example, a navigable association end (i.e., *Property*) is *owned* by the opposite *Class* through *ownedAttribute*, and the *Association* can still access the *Property* through its *memberEnd* at the same time. This indicates that a part is allowed to be exposed outside of its owner's boundary. This relaxed constraint makes it difficult for Milanova's approach to verify whether the implementation satisfies the coincident lifetime requirement or not.

Guéhéneuc and Albin-Amiot [9] used a combination of static and dynamic analysis to recover compositions from a Java program. Their definition does not well capture the notion of compositions that arise in many programs in practice. For instance, their exclusivity property for a composition requires that a part may not even participate in relationships with any other objects except its owner. We argue that it is perfectly legal for a part to be *linked* with other objects, as long as these *links* do not represent compositions. For example, it is common for a part to reference some other objects in its implementation. Another disadvantage of their approach is that the composition recovery is based on particular executions of a system, which has the well-known code coverage problem and results in imprecisely recovered compositions.

Work in [7] and [14] applied a dynamic approach to checking a program's behavior against some assertions (i.e., preconditions, postconditions, and class invariants) expressed in the Object Constraint Language (OCL). However, their techniques only discuss how to check OCL assertions in general, without addressing the verification of the constraints imposed by associations and compositions, which are two of the most important relationships in UML. Moreover, some constraints such as *exclusivity* and *lifetime* cannot be directly expressed in OCL without precisely defining their semantics.

In this paper, we focus on precisely defining the run-

Fifth International Workshop on Dynamic Analysis (WODA'07)
0-7695-2963-1/07 $20.00 © 2007 IEEE

time implications of these two relationships in terms of four class invariants: *navigability*, *multiplicity*, *exclusivity*, and *lifetime*, and verifying whether these invariants are satisfied or not during a program's execution. Our approach instruments Java class files to hook up the event notification mechanism into certain points in methods. The events generated during a program's execution are processed by a collection of *event handlers*, which update our consistency checking model and verify the class invariants. We have developed a prototype consistency checking tool and evaluated it on third-party programs such as Eclipse UML2 [2] and NSUML [3]. The empirical results show that our approach can precisely and efficiently detect implementation or design flaws.

2. Definitions

An association specifies a semantic relationship that can occur between typed instances. The *navigability* and *multiplicity* properties of an association end impose important constraints on a runtime system. A navigable association end means that the instance(s) of the target class can be *efficiently* accessed by an instance of the source class. For a binary association, the *multiplicity* on the target end constrains the number of objects of the target class that may be associated with a single object of the source class. For brevity, we use the notation $A : X \rightarrow (y)Y$ to denote a unidirectional association from X to Y with association end y navigable, and we use the notation $A : X(x) \leftrightarrow (y)Y$ to denote a bidirectional association between X and Y with both ends navigable.

A composition is a special form of association and specifies a strong ownership between a whole (composite) and its parts. It signifies two additional constraints on a runtime system: *exclusivity* and *lifetime*. The *exclusivity* requires that a part be *owned* by at most one composite at a time. The *lifetime* requires that if a composite is *destroyed* (*deleted*), all of its parts be normally *destroyed* with it. Compositions are transitive and asymmetric, and therefore, the set of all composition relationships in a runtime system forms a forest of trees (i.e., ownership hierarchies) made of objects and composition *links* [18].

To precisely capture the runtime implications of the design-level associations (keep in mind that a composition is a special form of association), we first introduce three instance-level concepts: *link*, *own*, and *destroy*. We then define the runtime implications of associations based on these concepts in Section 3. In addition, we distinguish between two types of objects: application objects and container objects (containers, for brevity). An application object is an instance of a class in the class model, and a container is an instance of either *java.util.Collection* or *java.lang.Object[]*.

• Link

A *link* is an instance of an association. We classify *links* into *direct links* and *transitive links*. A *direct link* is a *link* between two application objects or one between an application object and a container. At runtime, a *direct link* can be created when:

1) An application object o holds a reference to o' through an instance field f' at some point during a program's execution, where o' is either an application object or a container. We use the notation $l : o \rightarrow (f')o'$ to represent such a *direct link*, or

2) An application object o is added to a container c at some point during a program's execution. That is, an application object is added to a collection through a method call on the collection, or is assigned to an array element. We use the notation $l : c \rightarrow o$ to represent such a *direct link*.

A *transitive link* $l : o \rightarrow (f')o'$ is derived from two *direct links*: $l : o \rightarrow (f')c$ and $l : c \rightarrow o'$. Typically, a single-valued association (multiplicity 1 or 0..1 at the target end) is declared as a simple field whose type is the target class, while a multi-valued association (multiplicity upper bound greater than 1 at the target end) is declared as an instance field of container type [18]. Thus, *direct links* between application objects are actual instances of design-level single-valued associations. However, *direct links* between application objects and containers usually do not represent any instance of design-level associations. Instead, *transitive links* between application objects are actual instances of design-level multi-valued associations. For example, Figure 1 shows that object x of class X has a *direct link* to a container c through field f, and the container c has two *direct links* to objects y_1 and y_2 of class Y (i.e., c contains y_1 and y_2), then two *transitive links* $l : x \rightarrow (f)y_1$ and $l : x \rightarrow (f)y_2$ can be derived.

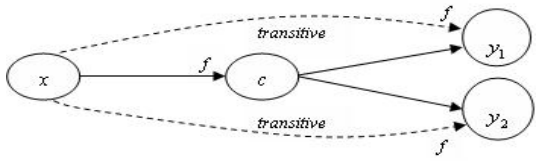

Figure 1. Transitive links between application objects

Whenever we mention a *link* between two application objects in the following text, we mean either a *direct* or a *transitive link* between them. Sometimes we use the phrase o accesses o' to denote that there is a *link* from application object o to application object o'.

• Own

An *own* relationship between two application objects is

0-7695-2963-1/07 $20.00

defined in terms of *links* (either *direct* or *transitive*). At runtime, application object *o owns* application object o' if there exists a *link* $l : o \rightarrow (f')o'$ or $l : o' \rightarrow (f)o$, which represents an instance of a composition in the class model, and if o and o' are the instances of composite and part classes, respectively. The *own* relationship is transitive - if o_1 *owns* o_2 and o_2 *owns* o_3, then we have o_1 *owns* o_3.

Note that we shall compare the runtime system with the class model in order to determine: 1) whether an object is an application object, 2) whether an object represents an instance of a composite or part class, and 3) whether a reference through an instance field represents an instance of a design-level association. Here, we assume that an object is an application object if the metaclass of the object has the same name as a model-level class. It follows that this object represents an instance of a composite (or part) if its model-level class is a composite (or part) one.

Typically, a unidirectional association is mapped to an instance field declared in the source class, and a bidirectional association is mapped to a pair of instance fields, each of which is declared in one class participating in the association [18]. Therefore, a runtime *link* may represent a design-level unidirectional association or half (one direction) of a bidirectional association. We assume that, in the source class, the name of a field implementing a design-level association is the same as the name of the target end of the association. Hence, if x and y are respectively instances of X and Y, then the *link* $l : x \rightarrow (f)y$ represents an instance of an association $A : X \rightarrow (f)Y$ or half of the association $A : X(f') \leftrightarrow (f)Y$. It follows that the runtime *links* $l : x \rightarrow (f)y$ and $l : y \rightarrow (f')x$ together represent an instance of the bidirectional association $A : X(f') \leftrightarrow (f)Y$ in the class model. A *link* $l : x \rightarrow (f)y$ is called a composition *link* if its design-level association is a composition (i.e., there is an *own* relationship between x and y).

• Destroy

Object *destruction* in this paper has a different meaning from the object garbage collection in the Java Virtual Machine (JVM) [11]. That is, *destroying* an object does not necessarily mean that the object has to be garbage collected. For the purpose of composition checking, we propose to use a special, public *destruction* method (e.g., *destroy()*) to signal the *deletion* of an object. A program implementing compositions should define a *destruction* method in all composite classes. The *destruction* method should implement the following semantics: after it is invoked on a composite object, the sub-tree rooted at this object should be detached from the ownership hierarchy it belongs to, and all incoming *links* from outside of the detached sub-tree should be deleted.

The proposed *destruction* method well captures the common intuitions about the composition's lifetime constraint, and has been adopted in some well-designed software libraries [3, 8, 2]. The invocation of the *destruction* method signals the start of a composite object's *deletion* process. After it completes, the object is considered to be *destroyed* and should no longer be *owned* by its owner. Moreover, no object within the sub-tree rooted at this composite can be accessed by any other object outside of the sub-tree, i.e., objects *owned* (including transitively *owned*) by the *destroyed* composite are considered to be *deleted* too. Note that the semantics of *destroy* does not prevent a *destroyed* object from being accessed by objects from within the detached sub-tree.

3. Runtime implications of associations

As mentioned in Section 2, multiplicity and navigability attach important information to an association end. Therefore, for a software system to conform to the constraints imposed by associations, at least the following two class invariants must hold at runtime.

1) Navigability Invariant

If there exists a *link* $l : o \rightarrow (f')o'$ at some point during the program's execution, there must exist a design-level association of the form: $A : C \rightarrow (f')C'$ or $A : C(f) \leftrightarrow (f')C'$, where o and o' are instances of C and C', respectively.

The navigability invariant states that a *link* must correspond to a bidirectional association or a unidirectional association with the same navigability. The rationale is that at runtime, o can always *efficiently* access o' through the *link* represented by its instance field f', and the corresponding association should be navigable at end f' as well.

2) Multiplicity Invariant

If object o of class C is connected to a set of objects of class C' through *links* instantiated from a single association $A : C \rightarrow (f')C'$ or $A : C(f) \leftrightarrow (f')C'$, then the size of this set must be between m and n, where m and n are the lower and upper bounds of the multiplicity at end f'.

The multiplicity invariant states that for a single object, the number of all objects connected to it through *links* instantiated from the same association must be within the range of the multiplicity on the target association end.

A composition signifies two additional invariants that a runtime system must hold:

3) Exclusivity Invariant

If object o *owns* object o', then no other object may *own* o' at the same time.

The exclusivity invariant implies that the upper bound of the multiplicity on a composition's owner end must not

be greater than 1, i.e., a part can be *owned* by at most one composite at any time. Note that the exclusivity invariant only means that a part cannot be directly *owned* by two composites, and does not prevent the part from being transitively *owned* by multiple composites at different levels in the ownership hierarchy.

4) Lifetime Invariant

If object o *owns* object o', then o' must be *destroyed* with o if o is *destroyed*.

Lifetime invariant requires that the parts have coincident lifetime with the composite, which has the sole responsibility for the *destruction* of its parts.

4. Runtime checking model - the object graph

Checking whether a runtime system conforms to the design-level association-related constraints is to verify the above four invariants during a program's execution. The verification is based on an evolving object graph, where nodes represent runtime objects and directed edges represent *direct links*. The object graph is similar to the ones proposed by Potter et al. [17] and [13], but is dynamically updated during the program's execution to reflect the current state of the runtime system. Below we introduce an example to illustrate how the object graph is updated on the fly.

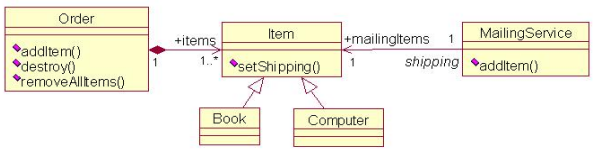

Figure 2. A simplified online shopping system

Figure 2 shows the class diagram of a simplified Online Shopping System (OSS). A composition between *Order* and *Item* specifies that a single item must belong to one order at a time and the item must be *deleted* with its order if the order is *deleted*. Each item is associated with a mailing service indicating how this item will be shipped. A possible implementation of the model is given below:

```
class Item {
    MailingService shipping;
    void setShipping(MailingService shipping) {
    1. this.shipping = shipping;}
}
class Order {
    List items;
    public Order() {
    2. items = new ArrayList();}
    void addItem(Item item) {
    3. items.add(item);}
    void removeAllItems() {
    4. items = null;}
    public void destroy() {
    5. removeAllItems();}
}
class MailingService {
    6. List mailingItems = new ArrayList();
    void addItem(Item item) {
    7. mailingItems.add(item);}
}
public class Application {
    public static void main(String[] args) {
    8. Order bookOrder = new Order("BookOrder");
    9. Book book = new Book("Book");
    10. bookOrder.addItem(book);
    11. Order dvdOrder = new Order("DVDOrder");
    12. dvdOrder.addItem(book);
    13. MailingService regMail = new MailingService("RegularMail");
    14. book.setShipping(regMail);
    15. regMail.addItem(book);
    16. regMail.addItem(new Computer("Laptop"));
    17. dvdOrder.removeAllItems();
    18. bookOrder.destroy(); }
}
```

Figure 3. A possible implementation of the OSS

One thing of particular interest is the *destroy()* method in the composite class *Order*, which clears its instance field *items* by calling *removeAllItems()*.

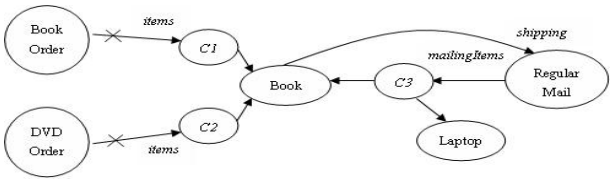

Figure 4. Object graph after Application.main()

Figure 4 illustrates the object graph representing the state of the runtime system upon the completion of *Application.main()*. The cross signs mark the deletion of edges. The object graph is updated as follows: a node representing an application object o is added to the graph after o is created; a node representing a container c is added to the graph when a *link* $l : o \rightarrow (f)c$ is created and if c is not on the graph yet; edges are added and deleted as corresponding *direct links* are created and broken, respectively. For example, after line 8 is executed, two nodes (*"BookOrder"* and *"C1"*) and an edge between them are added. *"C1"* represents the container accessed by *"BookOrder"* through field *items*. Although not shown on the object graph, *transitive links* between application objects are always derived immediately after new *direct links* are added to the graph. For example, *transitive link l: "BookOrder" → (items)"Book"* is derived after line 10 is executed. Figure 4 also illustrates the deletion of edges. For example, the execution of line 17 deletes the edge from *"DVDOrder"* to *"C2"*. Thus, the *transitive link* derived from it (*l: "DVDOrder" → (items)"Book"*) is deleted too.

0-7695-2963-1/07 $20.00

5. Runtime invariant checking

Whenever an object reaches an *observable* state (also known as *stable* state), the runtime invariant checking should be performed. According to Meyer [12], an *observable* state of an object can be reached in two cases: 1) immediately after the creation of the object, and 2) immediately after the execution of each exported routine of the object. In Java, an exported routine can be any non-private method of an object. Therefore, our solution is to check the four invariants after every invocation of a non-private instance method of an object (we deem a constructor as a non-private instance method although strictly speaking, it is not). One exception is the lifetime invariant, which should be checked only after a composite's *destruction* method completes.

1) Check Navigability Invariant

Navigability checking is performed for each *link* $l : o \rightarrow (f')o'$ created within a method. According to the navigability invariant, if there does not exist a corresponding design-level association $A : C \rightarrow (f')C'$ or $A : C(f) \leftrightarrow (f')C'$, the navigability invariant is violated at runtime. For instance, a violation is detected after line 1 is executed (called from line 14 in Figure 3). Line 1 creates the *link l: "Book" → (shipping)"RegularMail"*, whose corresponding association has an opposite navigability.

2) Check Multiplicity Invariant

Multiplicity checking is performed for each *link* $l : o \rightarrow (f')o'$ that is created or deleted within a method. If the *link* is an instance of the association $A : C \rightarrow (f')C'$ (or $A : C(f) \leftrightarrow (f')C'$), then 1) for object o, find all objects of C' connected to o through *links* instantiated from the association $A : C \rightarrow (f')C'$ (or $A : C(f) \leftrightarrow (f')C'$); a violation is detected if the total number of such objects is out of the range of the multiplicity at end f', and 2) apply the same checking process for o'. For example, a violation of multiplicity invariant is detected after line 7 is executed (called from line 16 in Figure 3). At this point, the runtime system reaches the state that the *"RegularMail"* has two *transitive links* pointing to both *"Book"* and *"Laptop"*, i.e., a single *MailingService* is associated with two *Items*.

3) Check Exclusivity Invariant

Exclusivity checking is performed for each composition *link* $l : o \rightarrow (f')o'$ created within a method as only adding a *link* may cause a violation of the exclusivity invariant. On the object graph, if the newly added edge $l : o \rightarrow (f')o'$ represents a composition *link*, a violation of the exclusivity invariant is detected if the part (either o or o') is connected by any other composition links. For example, a violation is detected after line 3 (called from line 12 in Figure 3) is executed. At this point, the runtime system reaches the state that both *"DVDOrder"* and *"BookOrder"* have a *transitive link* pointing to a single object *"Book"*, and that these two *transitive links* are instances of the composition $A : Order \rightarrow (items)Item$. This actually reveals an inconsistency between the design and its implementation - the implementation fails to prevent a single *Item* (i.e., *"Book"*) from being *owned* by two *Orders* (i.e., *"DVDOrder"* and *"BookOrder"*).

4) Check Lifetime Invariant

Our approach requires that the *deletion* of a composite be signaled by the invocation of its *destruction* method. Thus, the lifetime invariant should be verified only after a composite's *destruction* method completes. On the object graph, we check that whether there exist objects (parts), which were *owned* by the *destroyed* composite, and which are still being accessed by other objects outside of the detached subtree. For example, the *"BookOrder"* is considered to be *destroyed* after line 5 is executed (called from line 18 in Figure 3). Thus the part *"Book"* should also be *destroyed* in the sense that no object except its owner *"BookOrder"* could access it. However, the state of the object graph after line 5 shows that *"Book"* is still being accessed by *"RegularMail"* through a *transitive link l: "RegularMail" → (mailingItems)"Book"*. This reveals another inconsistency between the design and its implementation - the implementation fails to ensure a part to be *destroyed* with its owner.

6. Event notification and handling

To keep the object graph up-to-date and instantly detect any invariant violation during a program's execution, we need to instrument the program so that the consistency checking tool can be notified as *interesting* events occur. Within many possible events that can happen in a running program, we are only interested in five types of them: method entry, field assignment, collection call, array change, and method exit.

Our instrumentation is built on top of ASM [1]. The instrumentation for each type of event notification is implemented by a separate *instrumenter*, which inherits from a super class *MethodInstrumenter*. Before a class in the target program is loaded into the JVM at runtime, all *instrumenters* are applied to transform (instrument) the class. The order of the transformations is irrelevant.

During the instrumentation of each class, the *instrumenter* analyzes each instruction in a method and looks for the points, at which interesting events should be generated. After such a point is located, new bytecode instructions are inserted. When such points are reached at runtime, the new instructions notify the *event handlers*, which process the events upon receiving the notifications.

0-7695-2963-1/07 $20.00

1) Method Entry Notification and Handling

The *instrumenter* adds new bytecode instructions as the first ones to be executed within a method body. These new instructions notify the *method entry event handler*, which pushes an object *savedEdges* onto a stack called *allSavedEdges*. This stack is analogous to a call stack of a program and is maintained throughout the program's execution. The pushed object *savedEdges* is comparable to an activation record and stores two sets of edges: *newEdges* and *oldEdges*, which respectively store the newly created and deleted edges during the current method's execution.

2) Field Assignment Notification and Handling

The *instrumenter* looks for every *PUTFIELD f* instruction [11] corresponding to an instance field assignment and inserts new bytecode instructions to notify the *field assignment event handler* immediately after such instructions. For example, the constructor *Order()* of class *Order* looks like the following after instrumentation:

```
public Order() {
    items = new ArrayList();
    FieldAssingmentEventHandler.process(this, "items", items);
}
```

The *field assignment event handler* updates the object graph in the following manner: if an instance field f of object o is assigned a new value o_1, a new edge from o to o_1 is added to the graph and to the *newEdges* (node o may be added to the graph before the new edge is added. This happens when the assignment is executed within *o's* instance variable initializer [11] or constructors). At the same time, the edge from o to the old value o_2 through f is deleted from the graph and is stored into *oldEdges*.

3) Collection Call Notification and Handling

The contents of a standard collection (i.e., a collection defined in the Java Collections Framework [5]) can be changed through standard accessors (e.g., *add* or *remove*), whereas the elements of a user-defined collection may be changed by any other methods defined in the collection. As a result, we need to compute the difference of the contents of a collection before and after each method invocation on the collection. Therefore, the *instrumenter* looks for method invocation instructions of the form *INVOKEVIRTUAL C.m* and *INVOKEINTERFACE C.m* [11], where C represents a collection class. The *instrumenter* then inserts new bytecode instructions to notify the *collection invocation handler* immediately after these instructions. For example, the method *addItem()* of class *Order* looks like the following after instrumentation:

```
void addItem(Item item) { items.add(item);
    CollectionInvocationHandler.process(items);
}
```

Table 1. Invariant checking conditions

Checking Condition		Navigability	Multiplicity	Exclusivity
Add Link	Composition	√	√	√
	Non-composition	√	√	
Delete Link	Composition		√	
	Non-composition		√	

The *collection invocation handler* compares the current contents of the called collection with its old contents, which can be obtained from the object graph by traversing the edges originated from the collection. The difference is then computed. New edges are added to the object graph and to the *newEdges*, and old edges are deleted from the graph and stored into the *oldEdges*.

4) Array Change Notification and Handling

An array object may change its contents via assignments to its individual elements. The *instrumenter* looks for each *AASTORE i* instruction [11] and inserts new bytecode instructions to notify the *array change event handler* immediately after such instructions. The handling for array change is similar to that for collection calls, and thus is omitted here due to space limit.

5) Method Exit Notification and Handling

The *instrumenter* inserts new bytecode instructions immediately before every return instruction (such as *RETURN* [11]) to notify the *method exit event handler*. For example, the method *destroy()* of class *Order* looks like the following after instrumentation:

```
public void destroy() { removeAllItems();
    MethodExitEventHandler.process(this.getClass(), "destroy");
}
```

If the currently executing method is a constructor, the *method exit event handler* puts the newly created object on the object graph if it is not on the graph yet. As discussed in the beginning of Section 5, an object reaches a new observable state immediately after an instance method completes. Thus, the *method exit event handler* examines every edge in the *newEdges* and *oldEdges* and invokes different invariant checking algorithms depending on whether the edge is added or deleted, whether the edge represents a composition *link* or a non-composition *link*, and whether the current executing method is a *destruction* method.

Table 1 summarizes the checking conditions for each invariant upon the completion of a non-private instance method. Symbol √ denotes that the corresponding invariant is checked under some condition. Multiplicity is always checked for every added or deleted *link*; navigability is ver-

0-7695-2963-1/07 $20.00

Table 2. Runtime events and actions on the object graph

Event	Action on the Object Graph		
	Add Node	Add Edge	Delete Edge
Constructor Exit	√		
Field Assignment	√	√	√
Collection Call		√	√
Array Change		√	√

ified only when a *link* is added; exclusivity is checked only when the added *link* is a composition *link*. Note that the lifetime invariant is unconditionally verified when the executing method is a *destruction* method, and thus it does not appear in the table.

Table 2 summarizes the events that may cause updates to the object graph at runtime. Symbol √ indicates that the corresponding action may be taken by the event handler for some event. For example, the field assignment event may cause either an edge addition or an edge deletion, or both, depending on the field's values before and after the assignment. If the field's values before and after the assignment are non-null, then both actions are performed.

7. Conclusions

Dynamically monitoring violations of UML specified constraints is valuable in detecting inconsistencies between software implementations and their class models. In this paper, we presented an approach to checking the association-related constraints for a Java program in execution. We have developed a prototype consistency checking tool and evaluated it on third-party programs such as Eclipse UML2 and NSUML. The results showed that our approach can precisely and efficiently detect implementation flaws. For instance, our prototype tool revealed some Eclipse UML2 implementation bugs, which were confirmed by its developers. In some cases, the Eclipse UML2 1.x and 2.0.1 releases failed to delete incoming *links* from outside of a *destroyed* sub-tree of the ownership hierarchy. For example, the implementations failed to *destroy* a *Parameter* after its owning *Operation* was *destroyed*. A *Constraint* could still access the *Parameter* through *constrainedElement* after the *destruction* of the *Parameter's* owner, *Operation*. As another example, the prototype detected a violation of the multiplicity invariant in NSUML - an *Association* is allowed to be *linked* with only one *AssociationEnd* at runtime. However, the *link*'s corresponding association in the UML 1.3 specification [15] requires that an *Association* must be *linked* with at least two *AssociationEnds*.

Currently, we are enhancing our prototype consistency checking tool and plan to apply it to validating more real-world applications.

References

[1] ASM. http://asm.objectweb.org/.

[2] Eclipse UML2. Version 1.1.1. http://www.eclipse.org/uml2.

[3] NSUML. Version 0.4.20. http://nsuml.sourceforge.net.

[4] L. A. Barowski and J. H. C. II. Extraction and Use of Class Dependency Information for Java. In *Working Conference on Reverse Engineering*, pages 309–315, 2002.

[5] J. Bloch. Java Collections Framework. http://java.sun.com/docs/books/tutorial/collections/.

[6] D. Cooper, B. Khoo, B. R. von Konsky, and M. Robey. Java Implementation Verification Using Reverse Engineering. In *Australasian Conference on Computer Science*, pages 203–211, 2004.

[7] W. J. Dzidek, L. C. Briand, and Y. Labiche. Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In *Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends*, pages 10–19, 2005.

[8] A. Egyed. UML13 Interface Tool. http://www.alexander-egyed.com/tools/uml _ interface _ tool.html.

[9] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering Binary Class Relationships: Putting Icing on the UML Cake. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314, 2004.

[10] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. In *International Conference on Software Engineering*, pages 194–202, 1999.

[11] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, Second Edition. http://java.sun.com/docs/books/vmspec/.

[12] B. Meyer. Applying "Design by Contract". volume 25, pages 40–51, 1992.

[13] A. Milanova. Precise Identification of Composition Relationships for UML Class Diagrams. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 76–85, 2005.

[14] D. J. Murray and D. E. Parson. Automated Debugging in Java Using OCL and JDI. In *International Workshop on Automated Debugging*, pages 53–67, 2000.

[15] OMG. Unified Modeling Language Specification. Version 1.3, formal/00-03-01. http://www.omg.org/.

[16] OMG. Unified Modeling Language: Superstructure. Version 2.0, formal/05-07-04. http://www.omg.org/.

[17] J. Potter, J. Noble, and D. Clarke. The Ins and Outs of Objects. In *Australian Software Engineering Conference*, pages 80–89, 1998.

[18] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, Section Edition*. Addison-Wesley Professional, 2004.

[19] P. Tonella and A. Potrich. Reverse Engineering of the UML Class Diagram from C++ Code in Presence of Weakly Typed Containers. In *IEEE International Conference on Software Maintenance*, pages 376–385, 2001.

0-7695-2963-1/07 $20.00 © 2007 IEEE