Runtime Monitoring of Object Invariants with Guarantee

Madhu Gopinathan[1] and Sriram K. Rajamani[2]

[1] Indian Institute of Science,
gmadhu@csa.iisc.ernet.in,
[2] Microsoft Research India,
sriram@microsoft.com

Abstract. High level design decisions are never captured formally in programs and are often violated as programs evolve. In this paper, we focus on design decisions in which an object o works correctly only if another object p is in some specific states. Such decisions can be specified as the object invariant of o.

The invariant of o must hold when control is not inside any of o's methods (i.e. when o is in a steady state). From discussion forums on widely used APIs, it is clear that there are many instances where o's invariant is violated by the programmer inadvertently changing the state of p when o is in a steady state. Typically, o and p are objects exposed by the API, and the programmer (who is the user of the API), unaware of the dependency between o and p, calls a method of p in such a way that o's invariant is violated. The fact that the violation occurred is detected much later, when a method of o is called again, and it is difficult to determine exactly where such violations occur.

We propose a runtime verification scheme which guarantees that when o is in a steady state, any violation of o's invariant is detected exactly where it occurs. This is done by tracking dependencies automatically and validating whether a state change of an object p breaks the invariant of any object o that depends on p. We demonstrate that our tool InvCOP, which implements this scheme, can accurately pinpoint violations of invariants involving multiple objects that were reported in discussion forums on widely used APIs.

1 Introduction

Design decisions impose constraints on both the structure and behavior of the software. Typically, these decisions are described informally in comments embedded within code, or in documents. These documents are seldom updated as the software evolves. As a result, valuable design information is missing in most complex software. A promising approach to solve this problem is to capture design decisions formally as rules, and build tools that automatically enforce that programs obey these rules.

Data types are the only rules that are formally captured in programs, and enforced by programming languages. Over the past decade, we have witnessed

practical tools and type systems that extend this type of checking to allow stateful protocols on objects [1–3]. All these systems treat the state associated with each object independently. For example, they can check if every lock in the program is acquired and released in strict alternation, or if every file is opened before read, and then closed before the program exits. However, they are not capable of expressing rules that involve multiple inter-related objects. Since objects usually depend on other objects, such rules are common:

> *"... no object is an island. All objects stand in relationship to others, on whom they rely for services and control"* [4]

In this paper, we present a runtime verification approach for enforcing the following **inv-rule**: *The invariant of object o (which can refer to the state of other objects p) must hold when control is not inside any of o's methods.* The unique feature of our approach is that we track dependencies between objects automatically, and guarantee that violations of the **inv-rule** are detected exactly when they occur.

Example 1: Iterators for collection classes. Consider the Java code fragment below that uses an iterator to access the integers in a list sequentially.

```
1  //list is of type ArrayList<Integer>
2  //with integers 1,2,3 added
3  for(Iterator<Integer> i = list.iterator(); i.hasNext(); ) {
4    int v = i.next();
5    if(v == 1)
6      list.remove(v);
7    else
8      System.out.println(v);
9  }
```

On execution, a `ConcurrentModificationException` (CME) (the name is misleading as it occurs in single threaded programs also) is thrown at line 4. The API documentation for `ArrayList` [5] states the following:

> *If list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException.*

The `Iterator i` depends on the `list` to not change during iteration. This can be specified as the invariant of `Iterator`:

```
List myList = ..;
int expectedVersion = ..;

//object invariant of Iterator
public boolean Inv() {
  return myList != null &&
         expectedVersion == myList.version;
}
```

Since `list.remove` (line 6) removes an element from the list and changes `list.version` as a side-effect, the invariant of the iterator is violated at this point. However, this violation is detected only when the `next()` method is called at line 4. Hence CME is thrown at line 4.

Example 2: Statement and Connection. Consider the code below that uses JDBC(Java Database Connectivity) API to access a database.

```
1  Connection con = DriverManager.getConnection(..);
2  Statement stmt = con.createStatement();
3  ResultSet  rs1 = stmt.executeQuery("SELECT EMPNO
4                                      FROM EMPLOYEE");
5  ..
6  con.close();
7  ..
8  ResultSet rs2 = stmt.executeQuery("SELECT EMPNAME
9                                      FROM EMPLOYEE");
```

A statement depends on the connection used to create it for executing SQL statements. Closing a connection will invalidate any statement created by that connection. Calling any method of an invalid statement other than `isClosed` or `close` results in a `SQLException`. To avoid such errors, a connection must not be closed before closing any statement created by that connection. This can be specified as the invariant of `Statement`.

```
Connection connection = ..;
boolean isClosed = ..;

//object invariant of Statement
public boolean Inv() {
  return isClosed ||
    (connection != null && !connection.isClosed());
}
```

The invariant of `stmt` is violated at line 6 as `con` is closed before closing `stmt`. When the `executeQuery` method is called later, this is detected. Hence `SQLException` is thrown at line 8.

The **inv-rule** is difficult to enforce, since it requires keeping track of the state of related objects. In Example 1, a programmer may not be aware that iterator `i` depends on the list, and that changing the list will break the iterator's invariant. Similarly, in Example 2, a programmer may not be aware that closing the connection will break a statement's invariant. Such violations routinely occur in large programs [6–8]. Debugging such violations is hard. In Example 1, an exception is thrown at line 4, and the stack trace of the exception does not refer to line 6, which violated the rule. In Example 2, an exception is thrown at line 8, and the stack trace does not refer to line 6, which violated the rule.

Our goal is to enable providers of APIs to document the **inv-rule** precisely and provide a tool to help users of the API to detect exactly where violations occur in the user code. Thus, we can rely on the **inv-rule** being enforced in any program using the API.

We are not the first to consider rules involving multiple objects. Several "ownership" type systems have been invented to enable objects to own other objects they depend on [9, 10]. However, these require making changes to the programming language, and there is still a lot of debate on the pros and cons of various ownership type systems [11]. Also, program verification tools to check such invariants have been proposed, which force programmers to follow a particular methodology [12]. While this methodology works naturally for certain ownership structures, they need to be extended to handle cases where multiple objects depend on the same object [13] (as in Example 2 where multiple statements depend on the connection used to create them).

In this paper, we show how to enforce the **inv-rule**. The paper has two main contributions:

- We guarantee that in every run of a program, either a violation of the **inv-rule** is reported exactly where it occurs, or the run indeed satisfies the **inv-rule** (see Theorem 1 in Section 2.3 for a precise statement). This distinguishes our work from other runtime monitoring approaches to rule enforcement such as MOP [14], Tracematches [15] and JLo [16], where no such guarantees can be given if critical events from the program are missed by the monitor (see Section 4 for an example).
- Our approach is implemented in a tool called InvCOP. We demonstrate that rules involving objects exposed by the API are not violated by detecting usage errors previously reported in discussion forums on two commonly used APIs.

2 Approach

In this section, we explain the key features of our approach in stages, motivating the need for each feature. Consider Example 2 in Section 1. Our goal is to enforce the rule that in any program, a connection cannot be closed unless all the statements created using that connection are closed. We have seen that this can be specified as the object invariant of `Statement`.

```
public boolean Inv() {
  return isClosed ||
    (connection != null && !connection.isClosed());
}
```

Similarly, we can capture the dependency of iterator on list using the object invariant of iterator. Consider designing a reusable monitor object which reports an assertion violation if the **inv-rule** is violated. For every object that registers with the monitor, we require a side-effect free public method `boolean Inv()` returning a boolean, that checks the actual invariant (depending on the implementation of the object). To capture this requirement, we introduce the role *ObjWInv* (object with invariant).

```
role ObjWInv {
  boolean Inv();
}
```

For every object o of role *ObjWInv* (i.e. a subtype of *ObjWInv*), we add a boolean auxiliary field *inv*. Our goal is to ensure that for every object o of role *ObjWInv* in the program, whenever $o.inv$ is true, `o.Inv()` returns true. In the monitor, $o.inv$ is set to true only by using $CheckAndSetInv(o)$ that asserts $o.Inv()$ before setting $o.inv$ to true:

```
CheckAndSetInv(ObjWInv o) {
  assert o.Inv();
  o.inv = true;
}
```

The goal of the monitor is to report an assertion violation whenever a state change of p breaks the invariant of any o that depends on p. For this, the monitor must know the dependents of p. Therefore, we introduce another auxiliary field $ObjWInv.dependents$ of type *Set* of *ObjWInv*. To register an object o, the monitor's *Init* method must be called which initializes $o.inv$ to false and $o.dependents$ to empty set.

```
Init(ObjWInv o) {
  o.inv := false;
  o.dependents := nullset;
}
```

The monitor must be informed of dependencies (e.g. when a statement is created using a connection) by calling its *Add* method.

```
Add(ObjWInv o, ObjWInv p) {
  assert(o.inv = false);
  p.dependents.Add(o);
}
```

The monitor must be informed that o is in a steady state and $o.inv$ must be monitored by calling its *Start* method. Before executing a method of o, *Stop* must be called to indicate that $o.inv$ need not be monitored.

```
Start(ObjWInv o) {
  assert(o.inv = false);
  CheckAndSetInv(o);
}
```

```
Stop(ObjWInv o) {
  assert(o.inv = true);
  o.inv := false;
}
```

Whenever the state of an object p changes, we should check with the monitor by calling its *Validate* method. This method checks whether the state change of p breaks the invariant of any o that depends on p.

```
Validate(ObjWInv p) {
  for(o in p.dependents) {
    if(o.inv = true)
      CheckAndSetInv(o);
  }
}
```

Next, we need to instrument the program with appropriate calls to the monitor. Consider again, the JDBC user code given below. The calls to the monitor methods are shown in italics.

```
1  Connection con = DriverManager.getConnection(..);
2  Init(con); // register con
3
4  Statement stmt = con.createStatement();
5  Init(stmt); // register stmt
6  Add(con, stmt); // inform monitor that stmt depends on con
7  Start(stmt); //start monitoring stmt.inv
8  ..
9  Stop(stmt);  //stop monitoring stmt.inv
10 ResultSet  rs1 = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");
11 Start(stmt); //start monitoring again
12 ..
13 con.close();
14 Validate(con); // inform monitor that con's state changed
15 ..
16 ResultSet rs2 = stmt.executeQuery("SELECT NAME FROM EMPLOYEE");
```

With this added instrumentation, the call to *Validate* (line 14) reports an assertion violation as closing `con` breaks the invariant of `stmt` which is still open. Without the monitor, the error manifests subsequently, on line 16, when `stmt` is used. In this example, this is close to line 13, but in large programs the manifestation could be arbitrarily far away from the cause, resulting usually in long hours of debugging. With the monitor, we can detect the error at the point where **inv-rule** is violated (line 13).

However, there are two problems with the approach above:

1. Adding calls to monitor methods in the program creates a tight coupling between the monitor and the program bound to the rule. It is not easy to disable the monitor during deployment.
2. Errors can be missed if a call to an appropriate monitor method is omitted. For example, if the call to *Validate*(*con*) is omitted on line 13 above, then the error in the program is not detected by the monitor. The API programmer must ensure that the monitor knows about dependencies (by calling *Add*) and state changes of p are validated (by calling *Validate*). As new methods are added to p's class, the API programmer must ensure that appropriate calls to *Validate* are made. This process is error prone.

Sections 2.1 and 2.2 give solutions to problems 1 and 2.

2.1 Specifying bindings using AOP

Aspect oriented programming (AOP) [17] enables the various *concerns* (in this case, the JDBC specific code and the monitor specific code) to be specified separately. A description of the relationships of the two separate concerns enables

an AOP implementation such as AspectJ [18] to compose them together. Thus, if the relationship is correctly specified, then the appropriate monitor methods are implicitly invoked.

Any class in the program with a public method `boolean Inv()` can be bound to the role *ObjWInv* (i.e. it becomes a subtype of *ObjWInv*). The API programmer can bind the classes `Statement` and `Connection` as shown below (the binding below uses AspectJ syntax).

```
declare parents: Statement implements ObjWInv;
declare parents: Connection implements ObjWInv;
```

In AspectJ, a *join point* is an identifiable point, such as a call to a method or an assignment of a field, in the execution of a program. All join points have an associated context. For e.g., a method call has the context caller, target and arguments. These are the points at which the monitor specific code can be composed with the JDBC code. A *pointcut* is a set of join points. *Advice* is the code to be executed at the join points in a particular pointcut. At runtime, a *before* advice is triggered before the join point and an *after advice* after the join point.

The initialization of *ObjWInv* occurs before the constructor body of a class implementing *ObjWInv* executes. After this point, the auxiliary state of o is initialized by calling the monitor method $Init(o)$.

```
pointcut init(ObjWInv o) : initialization(ObjWInv.new(..))
                           && this(o);
after(ObjWInv o) : init(o) {
  Init(o);
}
```

After the field `Statement.connection` is set (during the construction of `Statement`), the target `Statement` o and the argument `Connection` p are collected and o is added as a dependent of p.

```
pointcut setConnection(ObjWInv o, ObjWInv p) :
                          set(Connection Statement.connection)
                          && target(o)
                          && args(p);
after(ObjWInv o, ObjWInv p) : setConnection(o,p) {
  Add(o,p);
}
```

After statement is created, the monitor is asked to start monitoring its invariant.

```
pointcut create() : call(Statement.new(..));
after() returning(ObjWInv o) : create() {
  Start(o);
}
```

Before closing the statement, the monitor is asked to stop monitoring its invariant.

```
pointcut stmtClose(ObjWInv o) : call(public void Statement.close())
                                && target(o);
before(ObjWInv o) : stmtClose(o) {
  Stop(o);
}
```

After closing the connection p, we must validate whether this change breaks the invariant of any statement o that depends on the connection p.

```
pointcut conClose(ObjWInv p) : call(public void Connection.close())
                               && target(p);
after(ObjWInv p) : conClose(p) {
  Validate(p);
}
```

It is easy to see that the `Iterator/List` code can be composed with the same monitor in a similar fashion. Mistakes can be made in the binding: suppose the pointcut `conClose` does not list the call to `Connection.close`. Then the monitor misses the critical event that a connection with an associated open statement is closed in the program. Other runtime monitoring approaches using AOP [14, 16] rely on the programmer to correctly specify all the events that create dependencies and change relevant state in the program. In section 2.2, we show how to improve upon this by automatically tracking dependencies (as `o.Inv()` executes) and calling *Validate* whenever a relevant state change occurs.

Note that in many cases, a default binding such that *Stop*(o) is called before and *Start*(o) is called after every public method execution on o suffices. Then with automatic dependency tracking and validation, the API programmer need not write a binding description at all. However, in some cases, a custom binding is needed (see section 3).

2.2 Automatic dependency tracking with validation

The key insight we have is that for any object o, the objects p on which o's invariant depends can be computed when `o.Inv()` executes using AOP techniques. Thus, if we compute these dependencies, we can check whether `o.Inv()` holds every time an object p that o depends on changes, and flag a violation of the **inv-rule** exactly where it occurs. In this section, we show how to track dependencies and validate relevant state changes automatically.

Definition 1. $(o, p, f) \in \mathcal{D}$ *iff the object invariant of* o *depends on the value of the field* $p.f$.

We must have $(o, p, f) \in \mathcal{D}$ iff the field $p.f$ is read during the last execution of `o.Inv()` (i.e. o is a dependent of p). Suppose the value of $p.f$ changes. It can potentially break the invariant of some o iff $(o, p, f) \in \mathcal{D}$ and $o.inv$ is true. *Validate* is invoked to check whether the change in value of $p.f$ breaks any such o's invariant.

The fields read during the execution of `o.Inv()` can be captured using AOP. We require that all such fields f must be either of a built-in type or of a subtype

of *ObjWInv*. This restriction is imposed so that we can register with the AOP execution environment for changes to these fields. We rely on the AOP execution environment to get a notification when the value of such a field f changes. More details on the implementation are given in section 3.

Consider Figure 1. In scenario 1, after the statement o is constructed, *Start*(o) is invoked. Since *Start* calls *CheckAndSetInv*(o), `o.Inv()` is called, and we compute all the fields $p.f$ that o depends upon. Therefore, $(o, p, \mathit{isClosed}) \in \mathcal{D}$. In scenario 2, after the field `Connection.isClosed` is set to false in `Connection.close`, *Validate*(p) is called. As *o.inv* is true and `o.Inv()` returns false, an assertion violation occurs. We now show that our approach always catches such errors subject to certain restrictions and assumptions.

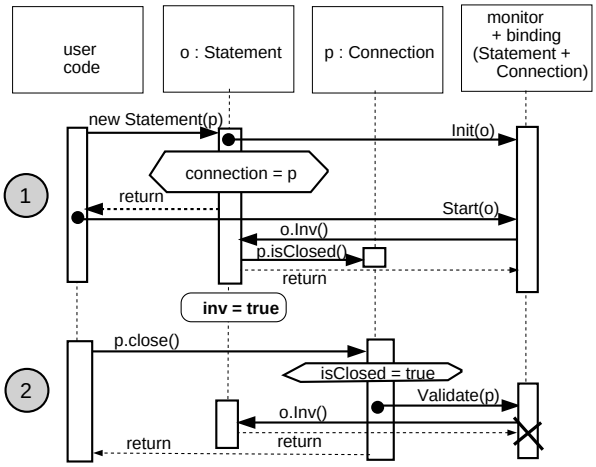

Fig. 1. Automatic dependency tracking with validation

2.3 Correctness

We consider sequential programs only. To summarize, the following restrictions on the program enable us to automatically track dependencies and validate state changes that may violate object invariants.

R1 Every field f read during the execution of `o.Inv()` must be either of a built-in type or of a subtype of *ObjWInv*.
R2 The execution of `o.Inv()` cannot modify any state.

These restrictions are checked by InvCOP using AOP, and if they are violated, an assertion violation is thrown.

The assumptions made on the AOP environment are:

A1 Every read access of a field $p.f$, where p is an object of type *ObjWInv*, can be detected.
A2 Every change of a field $p.f$, where p is an object of type *ObjWInv*, can be detected.
A3 The initialization of an object of type *ObjWInv* can be detected.

Assuming **A3**, we can ensure that every object o of type *ObjWInv* is registered with the monitor by calling *Init*(o). Note that *Init*(o) sets $o.inv$ to false. The following theorem relates the auxiliary field $o.inv$ to the actual invariant of the object `o.Inv()`.

Theorem 1. *Let r be any run of program P composed with the monitor using binding B. Suppose r does not have any assertion violations. Then, the following holds in all states of r:*

$$\forall o \in ObjWInv.(o.inv = true) \implies (o.Inv() = true)$$

Proof. The auxiliary field $o.inv$ is set to true only using *CheckAndSetInv*(o). Therefore, `o.Inv()` must have returned true. Let f be a field declared in a class P. Consider the assignment of a new value to $p.f$, where p is an instance of P. If this assignment violates the object invariant of o (i.e. `o.Inv()` now returns false and its previous execution returned true), then, the previous execution of `o.Inv()` must have accessed $p.f$. By assumption A1, we have $(o, p, f) \in \mathcal{D}$. By assumption A2, the assignment of $p.f$ is detected and *Validate*(p) is called. Since $o.inv$ is true and $(o, p, f) \in \mathcal{D}$, *CheckAndSetInv*($o$) is called. Since we assume that r does not have assertion violations, and *CheckAndSetInv*(o) calls `assert o.Inv()`, we have that `o.Inv() = true`.

3 Implementation

We have implemented the above approach in a tool called InvCOP. We first present the tool description followed by experimental results. The components of InvCOP are:

Monitor As we have discussed in section 2.
Depend Compute $\mathcal{D}$ during the execution of `o.Inv()`. Invoke the monitor method *Validate* when a state change is detected.
Aspect Generator Generate an aspect combining the above two components and a binding. In most cases, a default binding suffices. However, in certain cases (given below), the API programmer may need to provide a custom binding.

The Depend component uses AOP to to compute the dependency relation $\mathcal{D}$. The execution of `o.Inv()` is captured using a pointcut. Then using a control flow pointcut (cflow), any read operation of a field $p.f$ during the execution of `o.Inv()` can be captured. If a joinpoint specified by such a pointcut is reached, then (o, p, f) is added to $\mathcal{D}$. A call to `p.Inv()` during the execution of `o.Inv()` can be captured similarly to add (o, p, inv) to $\mathcal{D}$, i.e. o depends on $p.inv$.

The Aspect Generator generates an aspect A by combining the Monitor, Depend and binding. The binding is attached verbatim. For binding a class C to the role $A.ObjWInv$, AspectJ compiler modifies the inheritance hierarchy of C to introduce $A.ObjWInv$ as a parent. Currently, this is possible only if the byte code of C is under its control. Therefore, we cannot bind the collection classes in `java.util` to $ObjWInv$. Our prototype implementation uses proxy objects to keep track of the relationship between iterators and collection classes. At runtime, a singleton instance of the generated aspect is created in the virtual machine. This instance enforces the **inv-rule** by validating state changes of objects of type $A.ObjWInv$.

Custom Binding In the default binding, $Start(o)$ is invoked after the object o is constructed. Before the execution of every public method on o, $Stop(o)$ is invoked and after such an execution, $Start(o)$ is invoked again. However, in some cases, the API programmer may need to specify explicitly when *Start* and *Stop* are to be invoked. Consider the following example [12].

```
class T {
 public boolean Inv() {
    return 0 <= x && x < y;
 }

 public void method1() {
    x++;
    y++;
    //invoke method m on user object
    user.m(this,..);
    ..
 }
 public float method2() {
    return 1/(y-x);
 }
}

class User {
  public void m(T t,..) {
    //callback
    t.method2();
    ..
  }
}
```

If the API programmer allows the user to call back the method `T.method2` during the execution of `m`, then $Start(t)$ must be invoked before the call `user.m(this,..)`. Otherwise, $Stop(t)$ (called before executing `method2`) will report an assertion violation as $t.inv$ is false.

Experimental Results We first illustrate the difficulty faced by an API user using a real world scenario [7]. Figure 2 shows the usage of class `Document` in JDOM, a library for in-memory representation of XML documents.

A document iterator for navigating an XML document (in the form of a tree) uses a stack of list iterators where each list iterator is used for iterating over nodes at each level in the tree. An element in the tree is returned by the list iterator on top of the stack. If the user code calls `detach` on an element, then it is removed from the list of nodes at that level. An exception is thrown if user code invokes `detach` during iteration followed subsequently by `next` as shown in the stack trace below. The user code in the stack trace is shown in italics.

```
java.util.ConcurrentModificationException
        at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
        at java.util.AbstractList$Itr.next(Unknown Source)
        at org.jdom.DescendantIterator.next(DescendantIterator.java:134)
```

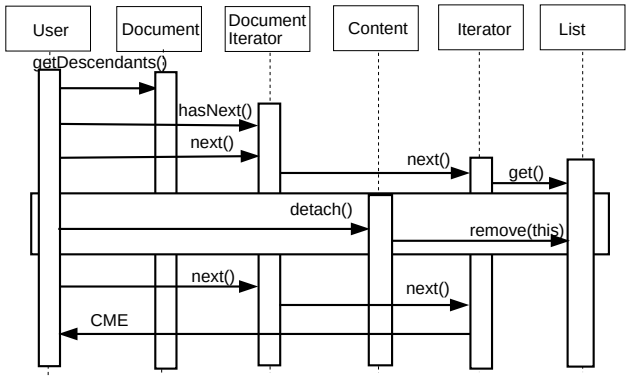

Fig. 2. Navigating a XML tree

```
        at org.jdom.FilterIterator.hasNext(FilterIterator.java:91)
        at OrderHandler.processOrder(OrderHandler.java:26)
```

From the above trace, it is not clear to the API user where exactly the invariant of iterator has been violated. After compiling with the aspect generated by InvCOP, the stack trace is as shown below.

```
java.lang.AssertionError: Invariant does not hold
        at rules.Inv_jdom.CheckAndSetInv(Inv_jdom.aj:122)
        ..
        at org.jdom.Element.removeContent(Element.java:885)
        at org.jdom.Content.detach(Content.java:91)
        at ItemHandler.processItem(OrderHandler.java:12)
        at OrderHandler.processOrder(OrderHandler.java:29)
```

This clearly points out that the user code *processItem* violated the iterator's invariant by calling `Content.detach`.

Table 1 shows some libraries for which we used InvCOP to detect **inv-rule** violations. Each scenario was modeled after usage violations reported in discussion forums. For each scenario, the columns show the total number of classes in the library and the number of classes bound to *ObjWInv*. We have already discussed the first scenario. Scenario 2 is based on an error report filed for MySQL JDBC library [8]. In scenario 3, user code first associates an implementation of `Key` with some value using a dictionary (implemented as a binary search tree). Then the key is modified violating the tree invariant.

With our prototype implementation, the time for each run with the generated aspect was 2-3 times that of the run without the aspect. However, this is insignificant compared to the amount of human effort spent in debugging these problems without a tool like InvCOP. Instead of documenting the reason for an exception in a FAQ (as in [6]), API users can be asked to use a tool such

Table 1. Detected **inv-rule** violations

API	Scenario	Total # of classes	Classes bound to *ObjWInv*
JDOM	Figure 2 [7]	69	IteratorProxy ListProxy
MySQL	[8]	95	Statement Connection
Binary Search Tree	[19]	5	BinarySearchTree Node Key

as InvCOP so that the violations of API rules can be detected quickly. Even if the API programmer has not formally captured all the API rules, as problems are reported, API rules can be captured incrementally. Once the violations have been found and fixed, the generated aspect can be removed during deployment.

4 Related Work

The SLAM toolkit [1] checks if C programs obey interface rules specified as state machines in the SLIC rule language. Powerful type systems have been designed to track a state machine as part of an object's type [2, 20, 3]. However, all these systems treat the state associated with each object independently. In this work, we focus on rules involving the states of multiple objects.

Contracts [21] identified behavioral compositions and obligations on participants as key to object oriented design. Recently, [22] has pointed out the need to enforce framework constraints (which typically involve multiple objects) so that plugin writers cannot violate them. These papers point to the need to automatically enforce constraints involving multiple objects in large programs.

Several "ownership" type systems have been invented to track dependencies between objects [9, 10]. The proposals in the literature differ in how they constrain programs: for example, some allow ownership transfer whereas some others do not. Program verification tools have been built to check if programmers follow particular programming methodologies [12]. When multiple objects depend on a shared object (many Statements may depend on the same Connection), the methodology needs to be extended [13]. Also, these systems do not work with existing programming languages.

JML [23] requires that an invariant must hold at the end of each constructor's execution, and at the beginning and end of all public methods. Our approach ensures that this is indeed the case during runtime.

MOP [14], Tracematches [15] and JLo [16] also use aspects for runtime verification. Consider the MOP specification (from [14]) for ensuring that a vector `v` is not modified when enumeration `e` is being used for enumerating the elements of the vector:

```
1 SafeEnum (Vector v, Enumeration+ e) {
2 [String location = "";]
```

```
3  event create<v,e>: end(call(Enumeration+.new(v,..))) with (e);
4  event updatesource<v>: end(call(* v.add*(..))) \/
5                         end(call(* v.remove*(..))) \/ ...
6                         {location = @LOC;}
7  event next<e>: begin(call(* e.nextElement()));
8  formula : create next* updatesource+ next
9  }
10 validation handler { System.out.println("Vector updated at "
11 + @MONITOR.location); }
```

In this MOP specification, a faulty pattern is specified using a formula which encodes incorrect sequences of events. After the event `create<v,e>` occurs, `e` depends on `v` to not change. The event `updatesource<v>` signals that the vector is modified. The formula `create next* updatesource+ next` specifies the faulty pattern: the enumeration is created, then vector is modified, followed by a `next` method call on the enumeration.

Suppose the specification of the event `updatesource<v>` inadvertently omits the method `v.remove()` (line 5 above). Then, an error similar to the one in Example 1 cannot be detected by MOP. In contrast, InvCOP does not require explicit specification of events that signal dependencies or state changes. With InvCOP, the programmer merely specifies that the enumerator depends on the vector's state. Whenever the state of the vector changes, automatic dependency tracking helps to check whether the invariant of the enumerator is violated. Thus, we believe that automatic dependency tracking is a useful feature that can be added to tools such as MOP to give guarantees such as the one offered by Theorem 1.

For us to track state changes of an object p that may affect the invariant of another object o, o must refer to p directly or indirectly. The AOP based monitoring approaches mentioned above do not place any such restrictions. However, the advantage of our approach is that we can enforce the **inv-rule** without the programmer having to list all methods that change object state and potentially break some other object's invariant.

5 Conclusion

We have presented an approach to formally capture design decisions which require an object o to constrain the state changes of another object p. We have also shown that our tool InvCOP guarantees to enforce such design decisions. Compared to other runtime verification approaches based on AOP, our approach reduces the specification burden on API programmers for the kind of design decisions we focus on in this paper. This is due to our novel dependency tracking and validation mechanism.

We have used our tool InvCOP to accurately pinpoint several usage violations that involved inter-related objects, reported in discussion forums on widely used APIs. Extending our work to concurrent programs, handling subclasses, and building a modular and scalable static analysis scheme for enforcing such design decisions require further research, and are beyond the scope of this paper.

References

1. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: POPL, ACM (January 2002) 1–3
2. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: PLDI, ACM (2001)
3. Chin, B., Markstrum, S., Millstein, T.: Semantic type qualifiers. In: PLDI, ACM (2005) 85–95
4. Beck, K., Cunningham, W.: A laboratory for teaching object-oriented thinking. In: OOPSLA. (1989) 1–6
5. `http://java.sun.com/j2se/1.5.0/docs/api/`
6. JDOM FAQ – `http://www.jdom.org/docs/faq.html#a0390`
7. `http://www.jdom.org/pipermail/jdom-interest/2005-March/014694.html`
8. `http://bugs.mysql.com/bug.php?id=2054`
9. Clarke, D.G., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA. (1998) 48–64
10. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL, ACM (2003) 213–223
11. Boyland, J.: Why we should not add readonly to java (yet). JOT **5**(5) (2006) 5–29
12. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. JOT **3**(6) (2004) 27–56
13. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: MPC. Springer-Verlag (2004) 54–84
14. Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: OOPSLA. (2007) 569–588
15. Avgustinov, P., Bodden, E., Hajiyev, E., Hendren, L.J., Lhoták, O., de Moor, O., Ongkingco, N., Sereni, D., Sittampalam, G., Tibble, J., Verbaere, M.: Aspects for trace monitoring. In: FATES/RV. (2006) 20–39
16. Stolz, V., Bodden, E.: Temporal assertions using aspectj. Electr. Notes Theor. Comput. Sci. **144**(4) (2006) 109–124
17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP. (1997) 220–242
18. AspectJ – `http://www.eclipse.org/aspectj/`
19. `http://www.ibm.com/developerworks/java/library/j-jtp02183.html`
20. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: PLDI, ACM (2002) 1–12
21. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: Specifying behavioural compositions in object-oriented systems. In: OOPSLA/ECOOP. (1990) 169–180
22. Jaspan, C., Aldrich, J.: Checking framework plugins. In: OOPSLA Companion. (2007) 795–796
23. Leavens, G., Cheon, Y.: Design by contract with jml (2003)