Enforcing Object Protocols by Combining Static and Runtime Analysis

Madhu Gopinathan
Indian Institute of Science
gmadhu@csa.iisc.ernet.in

Sriram K. Rajamani
Microsoft Research India
sriram@microsoft.com

Abstract

In this paper, we consider object protocols that constrain interactions between objects in a program. Several such protocols have been proposed in the literature [3, 9, 6, 5]. For many APIs (such as JDOM [23], JDBC [22]), API designers constrain how API clients interact with API objects. In practice, API clients violate such constraints, as evidenced by postings in discussion forums for these APIs. Thus, it is important that API designers specify constraints using appropriate object protocols and enforce them.

The goal of an object protocol is expressed as a protocol invariant. Fundamental properties such as ownership can be expressed as protocol invariants. We present a language, PROLANG, to specify object protocols along with their protocol invariants, and a tool, INVCOP++, to check if a program satisfies a protocol invariant. INVCOP++ separates the problem of checking if a protocol satisfies its protocol invariant (called *protocol correctness*), from the problem of checking if a program conforms to a protocol (called *program conformance*). The former is solved using static analysis, and the latter using runtime analysis. Due to this separation (1) errors made in protocol design are detected at a higher level of abstraction, independent of the program's source code, and (2) performance of conformance checking is improved as protocol correctness has been verified statically. We present theoretical guarantees about the way we combine static and runtime analysis, and empirical evidence that our tool INVCOP++ finds usage errors in widely used APIs. We also show that statically checking protocol correctness greatly optimizes the overhead of checking program conformance, thus enabling API clients to test whether their programs use the API as intended by the API designer.

OOPSLA '08 date, City.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.1 [*Requirements/Specifications*]: Tools; D.2.4 [*Software/Program Verification*]: Class invariants; D.2.5 [*Testing and Debugging*]: Monitors

General Terms Design, Languages, Reliability, Verification

Keywords Invariants, Aspect Oriented Programming, Program Verification

1. Introduction

Design decisions impose constraints on both the structure and behavior of software. Most practitioners of software engineering believe that there is value in capturing these design decisions as rules, and using tools to automatically enforce that programs follow these rules. In practice, data types are the only rules that are widely documented in programs and enforced by programming languages. Recent work has extended this approach to specify and check stateful protocols on objects [2, 13, 16, 8, 15]. These tools treat each object independently. Examples include checking if a lock is acquired and released in strict alternation, or checking if a file is opened before being read or written.

Object Invariants. The next challenge in this line of research is to specify and check rules that involve multiple inter-related objects. The simplest form of such rules are *object invariants*. For example, consider a class *Node* in a binary search tree with a *Key* field *k*, and fields *left* and *right* that point respectively to the left and right children of a node. The object invariant of *Node* states that $left.k < k$ and $k < right.k$. Furthermore, the invariant needs to hold recursively for the nodes pointed to by *left* and *right*. Thus, the invariant of *Node* depends on its fields *k*, *left* and *right*.

The fundamental difficulty with checking object invariants is that the invariant of an object o may depend on several objects $p_1, p_2, \ldots$. Thus, a change to p_i (possibly through aliases to p_i), may break the invariant of o, and there is no easy way to know the objects that depend on p_i. For example, if a client of binary search tree changes a key *k*, it might

break the invariant of a node that depends on *k*. Since *k* does not refer to *n*, such violations are hard to detect.

In recent work we have designed a tool INVCOP [18] to enforce object invariants at runtime. INVCOP tracks dependencies of objects automatically, and validates if a state change of object p breaks the invariant of any object o that depends on p. In our example, INVCOP automatically tracks that *Node n* depends on *Key k*. When *k* changes, it validates whether the change breaks the invariant of *n*. Thus, INVCOP guarantees that when an object o is in a steady state, any violation of o's invariant is detected exactly where it occurs.

Object Protocols. Object protocols specify generic constraints to structure interactions between objects. Several such protocols have been proposed in the literature [3, 9, 6, 5]. In order to check if programs follow these protocols, several techniques such as "ownership" type systems [9, 6], and program verification tools [4] have been proposed. The goal of an object protocol is expressed as a protocol invariant, which is parameterized over the object invariants of the participating objects.

As an example consider the *Representation Containment* protocol from [9]. The goal of this protocol is to ensure that if an object with an owner is accessed, then its owner must also have been accessed. Thus, an owner is assured that if the protocol is followed, then the objects it owns are not accessed without its knowledge. Additionally, if an object is not accessed, then its object invariant must hold.

Object protocols can be described operationally with: (1) auxiliary fields added to each object, (2) protocol methods that manipulate auxiliary fields, and (3) a protocol invariant over the auxiliary fields. For example, the *Representation Containment* protocol can be described by adding 2 auxiliary fields to each participating object: (1) $accessed$, which is true if o is being accessed, (2) $owner$, which points to the object that owns o. The protocol fields $accessed$ and $owner$ are manipulated by protocol methods (See Figure 3 in Section 3). The protocol invariant for this protocol is:

$$\begin{aligned} \forall o. \quad & (o.accessed \quad \Rightarrow \quad (o.owner = null \\ & \qquad\qquad \vee \ o.owner.accessed)) \\ & \wedge(\neg o.accessed \quad \Rightarrow \quad o.Inv()) \end{aligned}$$

Note that the protocol invariant is parameterized over object invariant $o.Inv()$ of participating objects. Each object type provides its own definition of the $Inv()$ method.

When API methods are called in a program, protocol methods are invoked using an appropriate binding. For example, when the key field of *Node n* is set to *Key k*, using a binding, the protocol method *Own(n,k)* (Figure 3, Section 6.2) is invoked which sets $k.owner$ to *n*. Suppose the root node is made the owner of all nodes in a search tree. The *Representation Containment* protocol governs how the auxiliary field $accessed$ of each node n in the tree changes and that whenever $n.accessed$ is false, the object invariant $n.Inv()$ holds.

Importance of Object Protocols. For many APIs (such as JDOM [23], JDBC [22]), API designers constrain how API clients interact with API objects. Several realistic design scenarios require a designer to constrain interactions between objects (see Sections 6.2.3,6.2.4). API clients commonly make mistakes in following these protocols (see Section 6). API designers need to design and customize object protocols according to the needs of the API (see Section 5). Therefore, we believe that it is important for API designers to state object protocols and enforce that API clients follow such protocols.

Checking Object Protocols. An obvious technique to check protocol invariants is to encode the object protocol with its auxiliary fields (for example, $accessed$ and $owner$ in the *Representation Containment* protocol) as part of the participating object, and encode the protocol invariant as the object invariant. Then, we can use a tool for checking object invariants (such as INVCOP) to check protocol invariants.

However, this technique has two major difficulties. First, errors can be made in the description of the protocol. It is unsatisfactory to detect such errors while checking if a particular program follows the protocol, since these are generic errors independent of the particular program being checked. It is desirable to detect such errors at the level of the protocol description itself (without bringing the program into the picture). Second, as our empirical results show, the overhead of checking if a program obeys a protocol invariant purely at runtime is unacceptably high (see Table 2 in Section 6). This overhead is due to a large number of changes to protocol fields, and validations that are consequently triggered by automated dependency tracking.

We solve both these problems using the following methodology. We specify the object protocol outside of the program using a simple language PROLANG. A PROLANG description of an object protocol consists of auxiliary fields, methods that manipulate auxiliary fields, and a protocol invariant. Each protocol method makes assumptions about the values of the auxiliary fields, and changes the values of the auxiliary fields (See Figure 3 in Section 3).

Our goal is to check if a given program M follows a protocol P with protocol invariant PI. Our methodology decomposes this check into two parts:

- First, we perform a *protocol correctness* check using static analysis (using automatic theorem provers). This checks if the protocol description of P in PROLANG, satisfies the protocol invariant PI.
- Next, we perform a *program conformance* check using runtime analysis. This checks if a program M satisfies the protocol P.

The protocol correctness check involves analyzing the protocol methods that manipulate the auxiliary fields, and checking if these methods satisfy the protocol invariant, while making assumptions specified in the bodies of these meth-

ods. The program conformance check involves only discharging these assumptions, and avoids checking the entire protocol invariant at runtime. Consequently, we are able to greatly optimize the runtime overheads during program conformance checking.

Our work has two main research contributions:

- Due to careful design of the protocol language PROLANG and the runtime system, we are able to formally prove that if the protocol verification(static check) passes, and program conformance(runtime check) passes for a run r, then the program indeed satisfies the protocol invariant for r (see Theorem 3 in Section 4). Thus, violations of protocol invariants are guaranteed to be detected exactly where they occur.
- We demonstrate that our language PROLANG is rich enough to express several object protocols from published literature (Section 5), and that the separation of concerns greatly reduces the overhead of checking program conformance (Section 6). We also demonstrate that by binding publicly available APIs with object protocols, we can detect usage errors reported in discussion forums for commonly used APIs (Section 6).

Benefits to API designers. INVCOP++ enables API designers to create a library of verified object protocols, and bind API classes or interfaces to appropriate protocols. Design mistakes in protocols are detected statically. A protocol description in PROLANG together with a binding to API classes provides precise documentation on how the API is to be used.

Benefits to API clients. INVCOP++ enables API clients to detect API protocol violations at runtime, exactly where they occur with no extra effort on their part (assuming that the API designer has included PROLANG descriptions of the object protocols governing API usage, bound to the API). By verifying protocols statically, the overhead in checking program conformance is significantly reduced.

Limitations. We are aware of two main limitations of our work:

- While our protocol correctness check is static and complete, our program conformance check is dynamic and incomplete. That is, we are able to guarantee program correctness only for the runs that we check, and we are unable to show that a program satisfies a given protocol invariant for all runs. This is the price we pay for flexibility. Unlike earlier approaches to object protocol specification, which force people to use particular ownership type systems, or particular pre-defined object protocols, we allow different parts of the API to use different protocols. We also allow API designers to *fine-tune* object protocols according to their requirements (see Section 5).
- Our techniques work only for sequential programs, and further work needs to be done to extend this to concurrent programs.

```
interface Key{
  //this < k?
  boolean Less(Key k);
}

class Node {
 Key k;
 Node left;
 Node right;

 Node(Key key) { k = key };

 //object invariant of this node
 boolean Inv(){
  return((left==null || left.k.Less(k) && left.Inv()) &&
         (right==null || k.Less(right.k) && right.Inv()));
 }
}

class BinarySearchTree {
 Node root;

 //insert a key
 void Insert(Key k) {
  //code to create a new node
  //and insert k
  ..
  assert root.Inv();
 }

 boolean Inv() {
   return (root==null || root.Inv());
 }
}
```

Figure 1. Binary Search Tree

Outline. The remainder of this paper is organized as follows. To make this paper self-contained, Section 2 reviews the core ideas behind INVCOP. Section 3 presents our protocol description language PROLANG. Section 4 describes the static and runtime analyses performed by our new tool INVCOP++, and states the formal guarantees behind the combination of static and runtime analysis. Section 5 illustrates that several object protocols from published literature can be modeled using PROLANG, and hence consequently verified and enforced using INVCOP++. Section 6 presents empirical results from running INVCOP++ on several APIs available in the public domain. Section 7 compares our work with related work, and Section 8 concludes the paper.

2. Background

In this section, we briefly explain the core ideas behind INVCOP [18].

Consider the class `Node` in Figure 1 that is used to construct a binary search tree. The invariant of `Node` is expressed by the `Inv()` method which states that if the left child is not null, then its key must be less than this node's key, and the tree rooted at the left child must be a valid binary search tree, i.e. `left.Inv()` must hold. A similar condition must hold if the right child is not null.

Consider the following client code that uses the BinarySearchTree API:

```
1  class IntKey implements Key {
2   int x;
3
4   boolean Less(Key k) {
5     IntKey o = (IntKey)k;
6     return x < o.x;
7   }
8  }
9
10 Key k1 = new IntKey(10);
11 Key k2 = new IntKey(5);
12 Key k3 = new IntKey(15);
13
14 BinarySearchTree bst = ..;
15 bst.Insert(k1);
16 bst.Insert(k2);
17 bst.Insert(k3);
18 //now, we have a tree with k1 as the key
19 //of the root node, k2 the key of its left
20 //child and k3 the key of the right child
21
22 //the following statement breaks the invariant of root
23 k2.x = 20;
24 ...
25 //assertion fails in bst.Insert
26 bst.Insert(new IntKey(..));
```

In the above example, the invariant of `bst` is violated by the statement `k2.x = 20` (line 23), but goes undetected, since there is no way for the runtime to know that an update to `k2.x` breaks the invariant of `bst`. Later in line 26, when `bst.Insert` is called and `root.Inv()` is checked, this violation is detected. The goal of INVCOP is to detect such violations exactly where they occur. Although it is not a good practice to insert mutable keys, API client programmers make such mistakes and it is difficult to find the root cause of the violation in large programs [10, 11].

INVCOP uses runtime verification to enforce the following **inv-rule**: *The invariant of an object o (which can refer to the state of other objects p) must hold when control is outside of any of o's methods.* The unique feature of INVCOP is that it tracks object dependencies automatically. In this case, after inserting the key `k3`, during the execution of `root.Inv()`, it is automatically detected that `root` depends upon keys `k1`,`k2` and `k3` since these fields are referred in `root.Inv()`. When `k2` is changed (line 23), INVCOP validates this change by asserting `root.Inv()`. Using such dependency tracking, INVCOP guarantees that violations of the **inv-rule** are detected exactly when they occur.

INVCOP assigns the *role ObjWInv* (object with invariant) to participating objects. This role has a boolean field inv, a side effect free function `Inv()` that returns a boolean, and a set $dependents$ of objects that depend on an object with this role.

```
role ObjWInv{
  boolean inv;
  boolean Inv();
  Set<ObjWInv> dependents;
}
```

The participating objects are required to provide only the `Inv()` function (as in `Node` in Figure 1). The fields inv and $dependents$ are automatically managed by INVCOP. INVCOP must ensure that whenever $o.inv$ is true, the function `o.Inv()` returns true in a program. By default, INVCOP sets $o.inv$ to true when control exits public methods of o, and sets $o.inv$ to false when control enters public methods of o. In the case of reentrant code, this can be modified as desired. INVCOP sets $o.inv$ to true only after asserting that `o.Inv()` returns true. In addition, during the execution of `o.Inv()`, INVCOP monitors all objects p that are accessed by `o.Inv()` and adds o to $p.dependents$. When p is modified, INVCOP automatically checks that for every $o \in p.dependents$, if $o.inv$ is true, then `o.Inv()` indeed holds. If `o.Inv()` does not hold, then a violation of o's invariant has been detected exactly where it occurred.

INVCOP offers the following correctness guarantee. A proof of this theorem can be found in [18].

Theorem 1. *Let r be any run of a program P which is run with* INVCOP*'s runtime. Suppose* INVCOP *does not raise any runtime violations during r. then, the following holds in all states of r:*

$$\forall o \in ObjWInv.(o.inv = true) \Rightarrow (o.\texttt{Inv}() = true)$$

3. Object Protocols

Recall the *Representation Containment* protocol from Section 1. This protocol ensures that every object o with an owner is accessed only with the knowledge of o's owner, and for every object o, its object invariant holds when it is not accessed. As we stated before, the protocol adds two auxiliary fields $accessed$ and $owner$ for each object, and the protocol invariant is:

$$\begin{array}{lll} \forall o. \quad (o.accessed & \Rightarrow & (o.owner = null \\ & & \quad \vee \ o.owner.accessed)) \\ \quad \wedge(\neg o.accessed & \Rightarrow & o.Inv()) \end{array}$$

Suppose we want a binary search tree to behave according to this protocol. We could treat the protocol's auxiliary fields as object fields, encode the above invariant as the object invariant, and use INVCOP to ensure that the object invariant holds. This scheme is very inefficient. To see why, recall the class `Node` class from Section 2. The protocol can be encoded in the class `Node` as follows:

```
class Node : ObjWInv {
 //program fields
 Key k;
 Node left;
 Node right;

 //protocol fields
 boolean accessed;
 ObjWInv owner;
```

```
//protocol invariant encoded as object invariant
boolean Inv(){
 return((!accessed ||
         (owner==null || owner.accessed)) &&
        (accessed ||
         (left==null ||
            left.k.Less(k) && left.Inv()) &&
         (right==null ||
            k.Less(right.k) && right.Inv())));
 }
}
```

Note that the encoded object invariant of `Node` is obtained by substituting the original object invariant of `Node` for `o.Inv()` in the protocol invariant (we have also rewritten implications using disjunctions). If we attempt to use INVCOP to enforce the object invariant of `Node`, it suffers serious performance problems as changes to the protocol fields need to be tracked in addition to changes to program fields, and additionally, changes to protocol fields need to be validated by nodes that depend on them.

Consider a tree with 100,000 nodes. Assume that the root owns all the nodes in the tree. This is in conformance with the protocol, since the root (owner) is accessed before any node is accessed. However, if the protocol is encoded into the program, the `Inv()` function for each node depends on root, since the `Inv` function mentions `owner.accessed` and the root is the owner for all nodes in the tree. This dependency is automatically tracked by INVCOP. When the tree is traversed, the root node is accessed first and its *accessed* field is set to true. INVCOP needs to validate this change with every node in the tree since every node in the tree depends on the `accessed` field of the root! As our empirical results show (see Section 6, Table 2, scenarios S3 and S4), this encoding is very inefficient due to a large number of invariant validations triggered by automatic dependency tracking.

Additionally, when the object protocol is encoded as part of a program, mistakes could be made. Such errors at the protocol level can be detected only when a protocol is applied to a specific program, which is unacceptable. Therefore, we must detect errors in the protocol description at a higher level of abstraction.

3.1 A language to specify object protocols

We have designed a language PROLANG to describe object protocols separately from programs, and solve both the problems mentioned above. A PROLANG description of a protocol looks like a class declaration with (1) auxiliary field declarations, (2) method declarations, and (3) a protocol invariant specification. However, the meaning of a auxiliary field declarations, method declarations, and protocol invariant are quite different from the usual notions associated with a class. Intuitively, the auxiliary field declarations add auxiliary state to *all* objects, method declarations manipulate auxiliary states of *multiple* objects, and protocol invariants are

$$
\begin{array}{rcl}
P & ::= & \textbf{protocol}\ pn\ body \\
body & ::= & \{auxfld^{+}\ method^{+}\ invariant\} \\
auxfld & ::= & t\ \textbf{ObjWInv}.fn\ := c \\
method & ::= & mn\ (arg^{+})\ \{s\} \\
arg & ::= & \textbf{ObjWInv}\ o \mid \textbf{Object}[]\ p \\
s & ::= & \textbf{assume}\ le | s; s \\
 & & \mid \textbf{if}(o.fn)\{s\}\ \textbf{else}\{s\} \\
 & & \mid \textbf{for}(p\ \textbf{in}\ o.sfn)\{s\} \\
 & & \mid o.fn := v \mid CheckAndSetInv(o) \\
 & & \mid o.sfn.Add(p) \mid o.sfn.Remove(p) \\
invariant & ::= & (\forall o : \textbf{ObjWInv} :: qle) \\
t & ::= & \textbf{boolean} \mid \textbf{ObjWInv} \mid \text{enum type} \\
 & & \mid \textbf{Set}\langle\textbf{ObjWInv}\rangle \\
c & ::= & \textbf{true} \mid \textbf{false} \mid \textbf{null} \mid \text{enum constant} \\
v & ::= & c \mid e \\
e & ::= & o \mid o.fn \mid o.prn() \mid o.prn(\textbf{Object}[]\ p) \\
le & ::= & \text{logical expression} \\
qle & ::= & \text{quantified le that has inv} \\
 & & \\
pn & \in & \text{protocol names} \\
fn & \in & \text{aux. field names of any type} \\
sfn & \in & \text{aux. field names of Set type} \\
mn & \in & \text{method names} \\
prn & \in & \text{predicate names} \\
o, p & \in & \text{variable names}
\end{array}
$$

Figure 2. Syntax of PROLANG

inductive invariants over the auxiliary state of *all* objects. Figure 2 shows the grammar for PROLANG. Below, we give informal descriptions of various parts of the grammar, followed by an example.

Auxiliary field declarations. Object protocols are specified by adding auxiliary fields to each object which is assigned the role $ObjWInv$, in addition to the fields we introduced in Section 2. In particular, the boolean inv exists by default, and can be manipulated by the protocol methods. The set $dependents$ used by dependency tracking also exists. This field cannot be accessed by the protocol methods.

Method declarations. Method declarations take one or more objects as arguments, make assumptions about the protocol state, and change the protocol state. There are two special methods: (1) The $Init(o)$ method takes an object of type $ObjWInv$ and is called every time an object o with subtype $ObjWInv$ is initialized. (2) The $Validate(o)$ method also takes an object of type $ObjWInv$. Whenever an object

p changes, INVCOP++ automatically calls $Validate(o)$ for every o whose `Inv()` method depends on p.

There are two restrictions on coding the methods. First, we stipulate that $o.inv$ can be set to true only by using $CheckAndSetInv(o)$, which asserts `o.Inv()` before setting $o.inv$ to true:

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

Next, we require that every code path in the method body for $Validate(o)$ must either call $CheckAndSetInv(o)$ or set $o.inv$ to false.

Protocol invariant declaration. The protocol invariant needs to be universally quantified over all objects of type $ObjWInv$. i.e., of the form $\forall o \in ObjWInv.\ \varphi(o)$.

Example. Figure 3 shows the *Representation Containment* protocol [9] specified using PROLANG. Note that this specification separates the protocol from the program unlike our earlier encoding, which mixed the protocol description with the program.

This protocol adds two auxiliary fields for every object that is assigned the role $ObjWInv$: (1) a boolean field $accessed$, and (2) a reference field $owner$. In addition, the boolean field inv is available to the protocol methods.

The protocol description specifies 5 methods: $Init$, $Validate$, Own, $Access$ and $Done$. Recall the restriction that every code path in $Validate(o)$ needs to either call $CheckAndSetInv(o)$ or set $o.inv$ to false. Note that the definition of $Validate(o)$ in Figure 3 indeed satisfies this restriction.

The protocol is associated with a program through a *binding* using Aspect Oriented Programming (AOP) [25]. Using such a binding, the method $Own(o, p)$ is invoked when o must own p, the method $Access(o)$ is invoked before a method call on o, and the method $Done(o)$ is invoked after a method call on o. For the binary search tree example, after setting the key field `k` of a node `n`, $Own(n, k)$ must be invoked. Before any public method call on any object that implements `Key`, $Access$ must be invoked and after the call, $Done$ must be invoked. A sample binding for this example is shown in Section 6.2.

Finally, note that the object protocol invariant is also part of the protocol description in Figure 3.

```
1  protocol RepresentationContainment {
2
3    // boolean ObjWInv.inv exists by default
4    boolean ObjWInv.accessed;
5    ObjWInv ObjWInv.owner;
6
7    Init(ObjWInv o) {
8      o.inv := false;
9      o.accessed := true;
10     o.owner := null;
11   }
12
13   Validate(ObjWInv o){
14     assume(o.accessed = true);
15     o.inv := false;
16   }
17
18   Own(ObjWInv o, ObjWInv p) {
19     assume(o.accessed = true);
20     assume(p.owner = null ||
21            p.owner = o);
22     p.owner := o;
23   }
24
25   Access(ObjWInv o) {
26     assume(o.accessed = false);
27     assume(o.owner = null ||
28            o.owner.accessed = true);
29     o.accessed := true;
30   }
31
32   Done(ObjWInv o) {
33     assume(o.accessed = true);
34     CheckAndSetInv(o);
35     o.accessed := false;
36   }
37
38   invariant(forall o : ObjWInv ::
39       (o.accessed  ==> o.owner = null ||
40                        o.owner.accessed) &&
41       (!o.accessed ==> o.inv = true));
42
43 }
```

Figure 3. *Representation Containment* Protocol

4. Analysis

Given an object oriented program and an object protocol with a protocol invariant expressed in PROLANG, we would like to ensure that the program satisfies the protocol invariant. Recall our first attempt to do this in Section 3, by encoding the protocol into the program, and encoding the protocol invariant as the object invariant, and the difficulties faced with such an approach.

We separate this verification problem into two parts.

- *Protocol correctness*: check if the object protocol P satisfies its protocol invariant PI.
- *Program conformance*: check if the program M conforms to the protocol P.

Our new tool INVCOP++ solves the protocol correctness problem using static analysis, and the protocol conformance problem using runtime analysis. We describe the two analyses below, and formally state and prove that the two analyses together ensure that the program M indeed satisfies the protocol invariant.

4.1 Protocol correctness.

We assume that the object protocol P is specified in PROLANG. Using automatic theorem proving (our implementation uses Simplify [14]) we check whether P satisfies

its protocol invariant PI. The object protocol P is considered as a state transition system of an unbounded number of objects, with the initial state of each object specified by executing the $Init$ method, and subsequent states obtained by executing the other methods in the protocol. We wish to establish that the set of all possible states that can be reached by executing the methods of the protocol P satisfy the protocol invariant PI. Let PI be of the form $\forall o \cdot \ \varphi(o)$. For the base case, we need to show that the $Init$ method satisfies the protocol invariant. Let $Init(S)$ be a predicate that holds whenever S is the auxiliary state produced by running the $Init$ action. We need to show that:

$$\forall(S) \cdot Init(S) \ \Rightarrow \ S \models PI$$

For the inductive case, we need to show that for each method A in the protocol description, if we execute A from a state S_1 that satisfies PI to reach a state S_2, then S_2 satisfies PI:

$$\forall(S_1, S_2) \cdot S_1 \models PI \wedge A(S_1, S_2) \ \Rightarrow \ S_2 \models PI$$

Our tool, INVCOP++, automatically generates these proof obligations from the protocol description, and uses an automatic theorem prover to check these proof obligations. If the proof obligation is not valid, then the automatic theorem prover gives a counterexample, which indicates why the protocol does not satisfy the protocol invariant.

The following theorem states that our static analysis indeed establishes that a protocol satisfies its protocol invariant.

Theorem 2. *Let P be any* PROLANG *protocol with protocol invariant PI, such that the $Init$ action satisfies $\forall(S) \cdot Init(S) \ \Rightarrow \ S \models PI$ and every other protocol method A satisfies $\forall(S_1, S_2) \cdot S_1 \models PI \wedge A(S_1, S_2) \ \Rightarrow \ S_2 \models PI$. Then, for every reachable auxiliary state S such that S is obtained by successive execution of protocol methods, we have that $S \models PI$*

Proof. By induction over the sequence of protocol methods executed. □

Example. Recall the *Representation Containment* protocol description from Figure 3. Let $init(o)$ be a formula that holds whenever o is the result of executing the $Init$ method. In this example, $init(o)$ is given below:

$$\begin{aligned} init(o) \quad = \quad & o.inv = false \ \wedge o.accessed = true \ \wedge \\ & o.owner = null \end{aligned}$$

Note that $init(o)$ can be generated automatically from the body of $Init$ in Figure 3. The protocol invariant PI is given by $\forall o.\ \varphi(o)$, where

$$\begin{aligned} \varphi(o) = \forall o. \quad & (o.accessed \Rightarrow \quad & (o.owner = null \\ & & \vee o.owner.accessed)) \\ & \wedge(\neg o.accessed \Rightarrow \quad & o.inv) \end{aligned}$$

The proof obligation for the $Init$ method is thus given by:

$$\forall o \cdot \ \ init(o) \Rightarrow \varphi(o)$$

To prove that the $Done$ method preserves the protocol invariant, we proceed as follows. We wish to produce a formula $done(o_1, o_2)$ that holds whenever an object can transition from state o_1 to o_2 by executing the $Done$ method. Such a formula is given below:

$$\begin{aligned} done(o_1, o_2) \quad = \quad & o1.accessed \ \wedge \ o2.inv \\ & \wedge \neg o2.accessed \\ & \wedge o1.owner = o2.owner \end{aligned}$$

Again, we note that $done(o_1, o_2)$ can be generated automatically from the body of $Done$ in Figure 3. The proof obligation for the $Done$ method is given by:

$$\forall o_1, o_2 \cdot \ \ \varphi(o_1) \wedge done(o_1, o_2) \Rightarrow \varphi(o_2)$$

Suppose the protocol designer forgets to set $o.inv$ to true in the method $Done$, i.e. line 34 is missing in $Done$. Then the above implication does not hold as there exists an object o_2 such that $\neg o_2.accessed \ \wedge \neg o_2.inv$ which satisfies $\varphi(o_1) \wedge done(o_1, o_2) \wedge \neg\varphi(o_2)$. The proof obligation for the $Done$ method fails with a counterexample that shows such an object o_2.

4.2 Program conformance.

After verifying a protocol P as above, we use runtime analysis to check if the runs of a program M conform to the protocol P. INVCOP++ uses Aspect Oriented Programming (AOP) to bind the protocol P's methods to appropriate points in the program M. After creation of every object o with role $ObjWInv$, $Init(o)$ is called to initialize o's auxiliary state. If an object p changes, then INVCOP++ guarantees to invoke $Validate(o)$ for every o whose `Inv()` depends on p.

INVCOP++ converts the assume statements in the methods of the protocol description P to assertions and checks them at runtime. This ensures that the assumptions made by the static analysis to prove the protocol invariant are indeed discharged at runtime.

For any protocol invariant PI let $\hat{PI}$ be the formula obtained by replacing every positive occurrence of $o.inv$ with $o.\texttt{Inv}()$. We refer to $\hat{PI}$ as the *instantiated protocol invariant*, since the occurrences of $o.inv$ have been instantiated with actual invariants from the program M. For the protocol invariant PI in our example, $\hat{PI}$ is:

```
forall o : ObjWInv ::
      (o.accessed = true ==> o.owner = null ||
                    o.owner.accessed = true) &&
      (o.accessed = false ==> o.Inv() = true)
```

Our main theorem composes the results of the protocol correctness and the program conformance phases.

Theorem 3. *Consider any protocol P with protocol invariant PI and methods that satisfy conditions of Theorem 2. Let r be any run of program M superimposed with protocol P using some binding. Suppose r does not have any assertion violations. Then, in all states of r we have that the instantiated protocol invariant $\hat{PI}$ holds.*

Proof. Three sets of assumptions made by the static analysis phase are guaranteed to be discharged by the runtime analysis. First, the static analysis assumes that every object o's auxiliary state is initialized by the method $Init(o)$, and the runtime ensures that the method $Init(o)$ is indeed called for every object with the role $ObjWInv$. Second, the static analysis assumes conditions stated in the method body on the auxiliary state during the verification of each method. These assumptions are converted into assertions and checked by the runtime analysis. Finally, automated dependency tracking ensures that $Validate(o)$ is called whenever an object p that o depends on changes. Since every code path in $Validate(o)$ either establishes $o.\texttt{Inv}()$ or sets $o.inv$ to false, we are able to reuse Theorem 1 in Section 2, and ensure that for every run r, the instantiated protocol invariant $\hat{PI}$ holds if no assertion failure is detected at runtime. □

Our work employs an intricate interplay between static and runtime analysis. During static analysis (to check protocol correctness) we assume conditions stated in protocol method bodies, and prove the protocol invariant inductively. During runtime analysis (to check program conformance), we merely discharge these assumptions, without having to check the entire protocol invariant. However, this alone is not sufficient to get the guarantee obtained in Theorem 3, since the object invariant of o can be changed if some p that o depends on changes. In addition, we need the automatic dependency tracking and invalidation scheme from our earlier work [18], and the constraints on the protocol methods (see Section 3.1) to prove Theorem 3.

Performance and correctness benefits. Consider again, the binary tree example from Section 3. Recall that when the object protocol invariant was encoded in the program, every time the root node was accessed, the protocol invariant had to be checked for all the nodes of the tree due to dependency tracking. However, by describing the protocol separately using PROLANG, and verifying the object protocol invariant statically, INVCOP++ greatly reduces the number of checks that need to be done at runtime. Indeed, the protocol invariant is never directly checked at runtime, and only the local assumptions made in the method bodies of the protocol description are checked at runtime. As our empirical results show in Section 6, this greatly enhances the performance of the runtime analysis. Further, errors made in the protocol description are now detected during the static analysis phase itself, independent of the program on which the protocol is enforced, leading to correctness benefits.

5. Scenarios

In this section, we demonstrate the expressiveness of PROLANG by encoding various object protocols from published literature. We motivate the need for each protocol by presenting a scenario that cannot be handled by the protocols discussed prior to it. In Section 5.2, we show a scenario where we need to "fine-tune" a protocol from the literature to suit the needs of the particular API under consideration.

In addition to the *Representation Containment* protocol that we have discussed in Section 3, we consider three other protocols: (1) the *Privileged Reader* protocol [6] that enables an object o to grant another object read access to objects owned by o, (2) the *Boogie* protocol [3] that enables an object o to own another object p and later, transfer the ownership of p to a third object and (3) the *Friends* protocol [5] that enables several friends to constrain state changes of a granter object that they all depend on.

5.1 Privileged Reader

Suppose that the API designer of `BinarySearchTree` used the *Representation Containment* protocol such that a tree owns all its nodes and keys. Consider implementing an iterator for iterating over the nodes in a tree as shown below. For simplicity, we show the protocol methods invoked (in bold) in the program itself.

```
class NodeIterator {
  BinarySearchTree tree = ..;

  //preorder traversal
  Node Next() {
    //current points to
    //the node on top of
    //a stack of nodes
    ..
    Access(current);
    Push(current.left);
    Done(current);
    ..
    return current;
  }
}
```

The `Next` method needs to access the nodes directly without accessing the tree. The *Representation Containment* protocol is too restrictive for this scenario. Figure 4 shows an extension of this protocol called *Privileged Reader* [6]. This protocol enables `BinarySearchTree` to let `NodeIterator` read its nodes. The protocol invariant states that if an object o is accessed, then either it has no owner or its owner has been accessed or a privileged object has accessed o.

Instead of invoking $Access$, the newly added method $PAccess$ must be invoked before an iterator accesses a node. The protocol invariant is maintained even though the owner of a node (tree) is not accessed as $PAccess$ sets $paccessed$ to true.

```
protocol PrivilegedReader {
  //in addition to the fields in
  //representation containment protocol
  boolean ObjWInv.paccessed := false;

  //Methods not shown are the same as in
  //the representation containment protocol.
  Done(ObjWInv o) {
    assume(o.accessed = true);
    CheckAndSetInv(o);
    o.accessed := false;
    o.paccessed := false;
  }

  //PAccess is a new method
  PAccess(ObjWInv o) {
    assume(o.accessed = false);
    o.accessed := true;
    o.paccessed := true;
  }

  invariant(forall o : ObjWInv ::
   (o.accessed = true ==>
      (o.owner = null ||
       o.owner.accessed ||
       o.paccessed)) &&
   (o.accessed = false  ==> o.inv));
}
```

Figure 4. *Privileged Reader* Protocol.

5.2 Ownership Transfer

In the protocols that we have discussed, once an object o owns another object p, it never gives up ownership of p. The API designer of `BinarySearchTree` would like the keys inserted by the API client to be owned by a tree only when the keys are part of that tree. When a key is deleted from a tree, then it must give up ownership of that key. Figure 5 shows the `Boogie` protocol [3] that allows ownership transfer.

The auxiliary field $o.st$ keeps track of whether object o is valid, invalid, or committed. The objects owned by o are elements of the set $o.comp$. The protocol invariant states that if an object o is either valid or committed, then o's invariant holds and all objects p on which o depends are also committed.

The field $o.st$ is modified by the methods $Pack$ and $Unpack$. When an object o is packed, it transitions from invalid to valid state and all objects p owned by o are committed to o. The API designer must pack an object o after construction and after a public method execution.

When object o is unpacked, it transitions from valid to invalid state and all objects p owned by it become valid. The API designer must unpack o before a public method executes on o. Note that o can be unpacked only if it is not committed. If o is committed, then all objects which depend on o must be unpacked before unpacking o.

The method $Own(o, p)$ adds p to the set $o.comp$, i.e. o owns p. The method $Giveup(o, p)$ removes p from $o.comp$, i.e. o gives up ownership of p. Using this protocol, every node can own its children and the associated key. The tree

```
protocol Boogie {
  enum State {Invalid, Valid, Committed};
  State ObjWInv.st;
  Set<ObjWInv> ObjWInv.comp;

  Init(ObjWInv o) {
    //Body supplied by rule designer
    o.inv := false;
    o.st := State.Invalid;
    o.comp := nullset;
  }

  Validate(ObjWInv o) {
    assume(o.st = State.Invalid);
    o.inv := false;
  }

  Pack(ObjWInv o) {
    assume(o.st = State.Invalid);
    CheckAndSetInv(o);
    for(p in o.comp) {
      assume(p.st = State.Valid);
      p.st := State.Committed;
    }
    o.st := State.Valid;
  }

  Unpack(ObjWInv o) {
    assume(o.st = State.Valid);
    o.st := State.Invalid;
    for(p in o.comp)
      p.st := State.Valid;
  }

  //o owns p
  Own(ObjWInv o, ObjWInv p) {
    assume(o.st = State.Invalid);
    o.comp.Add(p);
  }

  //o gives up p
  Giveup(ObjWInv o, ObjWInv p) {
    assume(o.st = State.Invalid);
    o.comp.Remove(p);
  }

  //rule invariant
  invariant (forall o: ObjWInv ::
       (o.st = State.Invalid ||
        (o.inv && (for all p: ObjWInv ::
                    (p in o.comp ==>
                      p.st = State.Committed)))));
}
```

Figure 5. *Boogie* Protocol.

can own the root node. When a key is deleted from the tree, the node that owned it must give up ownership of the key as follows:

```
class BinarySearchTree {
 void Delete(Key k) {
  Unpack(this);
  //Locate the node z that owns k
  Node z = ..
  //Delete z from the tree
  ..
```

```
  Giveup(z,k);
  Pack(this);
 }
}
```

Conceptually, this protocol serves the purpose of the API designer, i.e. prevent keys that are part of the tree from being modified by an API client and give up ownership of the key when it is deleted. However, this protocol does not allow the `BinarySearchTree.Delete` method to be implemented as in [12]. Suppose that the tree `bst` contains three nodes `root`, `y` and `z` with `y` as the left child of `root` and `z` as the left child of `y` and the key to be deleted is owned by the node `z`.

The value of the st field for these objects are shown in Figure 6. The tree `bst` is invalid as it has already been unpacked. Therefore, the `root` is valid and the nodes `y` and `z` are committed. To delete node `z`, node `y` must be unpacked. However, to unpack `y`, all the nodes along the path from the root to the node `y` must be unpacked and later they must be packed again in the reverse order. Thus, this protocol as it is, does not permit a straightforward implementation of the `Delete` method.

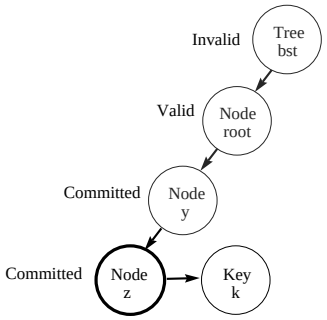

Figure 6. Problem deleting node `z`.

However, we can implement the `Delete` method *as is* by changing the ownership structure of this protocol, based on an idea in [9]. The tree can own all the nodes (as opposed to every node owning its children) and each node can own its associated key. Therefore, we modify the Boogie protocol to allow indirect references to owned objects as shown in Figure 7.

A new auxiliary field $owner$ is added. The protocol invariant has the additional condition that if an object is invalid, then its owner is either null or invalid. The new method $ToOwner(o, p)$ lets $o.owner$ to own p. This method must be invoked when a child field (`left` or `right`) of a node is set. As the node whose field is being set is owned by the tree, its child will also be owned by the tree even though the tree does not refer directly to the child node.

With the modified protocol, all nodes along the path from root to the node `y` are valid since the tree has already been unpacked. Therefore, `y` can be unpacked to delete node `z`. Thus, the `Delete` method need not be modified to accommodate the *Boogie* protocol. The modified protocol also allows a node to give up the key it owns when the key is deleted from the tree. This discussion illustrates the need for *mixing-and-matching* ideas from various protocols to *fine-tune* a protocol according to the requirements of the API designer.

```
protocol IndirectReference {
//new auxiliary field
//initialized to null
ObjWInv ObjWInv.owner;

ToOwner(ObjWInv o, ObjWInv p) {
  assume(o.owner != null
     && o.owner.st = State.Invalid);
  o.owner.comp.Add(p);
  p.owner := o.owner;
}

Unpack(ObjWInv o) {
  assume(o.owner = null
    || o.owner.st = State.Invalid);
  assume(o.st = State.Valid);
  o.st := State.Invalid;
  for(p in o.comp)
    p.st := State.Valid;
}

invariant (forall o : ObjWInv ::
  (o.st = State.Invalid ==>
                (o.owner = null ||
                 o.owner.st = State.Invalid))
  &&
  (o.st != State.Invalid ==>
    (o.inv && (p in o.comp ==>
            p.st = State.Committed))));
}
```

Figure 7. *Boogie* Protocol with Indirect Reference.

5.3 Friends Constrain Granter

Consider the classes `Connection` and `Statement` in the JDBC API [22]. The API designer must express the constraint that if any statement s created using a connection c is open, then the connection c must not be closed. Since several statements can share the same connection, this constraint cannot be enforced by letting a statement s own its connection c using one of the protocols above as all of them allow an object to have at most one owner.

The *Friends* protocol [5] shown in Figure 8 can be used by the JDBC API designer. The auxiliary field $granter$ points to the shared object on which constraints are imposed and the field $friends$ contains the objects which impose constraints on the granter. In this example, the $granter$ field of all statements s point to the connection c and the $friends$ field of c contains all statements s created using c. If the field ok of a friend is true, then that friend agrees to a proposed state change of the granter. The protocol invariant states that if a friend f is valid and its granter is not null, then the granter must know about f, i.e. $f \in f.granter.friends$ and

```
protocol Friends {
  enum State {Invalid, Valid};
  State ObjWInv.st;
  boolean ObjWInv.ok;
  ObjWInv ObjWInv.granter;
  Set<ObjWInv> ObjWInv.friends;

  Init(ObjWInv f) {
    f.inv := false;
    f.st := State.Invalid;
    f.ok := false;
    f.granter := null;
    f.friends := nullset;
  }

  Validate(ObjWInv f) {
   if(f.st = State.Valid)
     CheckAndSetInv(f);
   else
     f.inv := false;
  }

  Attach(ObjWInv g, ObjWInv f) {
    assume(g != null);
    assume(g.st = State.Invalid);
    assume(f.granter = null);
    f.granter = g;
    g.friends.Add(f);
  }

  Detach(ObjWInv g, ObjWInv f) {
    ..
  }

  UpdateGuard(ObjWInv g, Object[] args) {
    assume(g.st = State.Invalid);
    for(f in g.deps) {
      assume(f.granter = g);
      f.ok := f.OK(args);
      assume(f.st = State.Invalid || f.ok);
    }
  }

  Pack(ObjWInv f) {
    assume(f.st = State.Invalid);
    ..
    f.ok := true;
    f.st := Valid;
  }

  Unpack(ObjWInv o) {
    assume(o.st = State.Valid);
    o.st := State.Invalid;
  }

  invariant(forall f : ObjWInv ::
    f.st = State.Valid ==> f.inv &&
                  (f.granter = null ||
                  (f in f.granter.friends && f.ok)));

}
```

Figure 8. *Friends* Protocol.

f agreed to the last state change of the granter, i.e. $f.ok$ is true.

The API designer can invoke the methods $Attach$ and $Detach$ to add and remove friends to a granter. The auxiliary field ok of a friend f is set to true when the object f is packed. The API designer invokes $UpdateGuard$ before a state change of the granter. This method queries each friend using the method `OK`. If the friend f agrees to the change, then the field $f.ok$ is set to true, otherwise it is set to false.

Figure 9 shows how the JDBC API designer would use the *Friends* protocol. In scenario 1, a statement s created using a connection c is attached as a friend of the granter c. If the statement s is open, then it constrains updates of the field `Connection.closed` of the granter c such that `Connection.closed` is not set to true. In scenario 2, an API client closes the connection c without being aware that the statement s is still open. Before updating this field, the protocol method $UpdateGuard$ is invoked. This invokes the method `OK` on the friend s which returns false and the auxiliary field $s.ok$ is set to false. At runtime, this causes an assertion violation as the statement s is valid and not ok with the proposed state change of connection c.

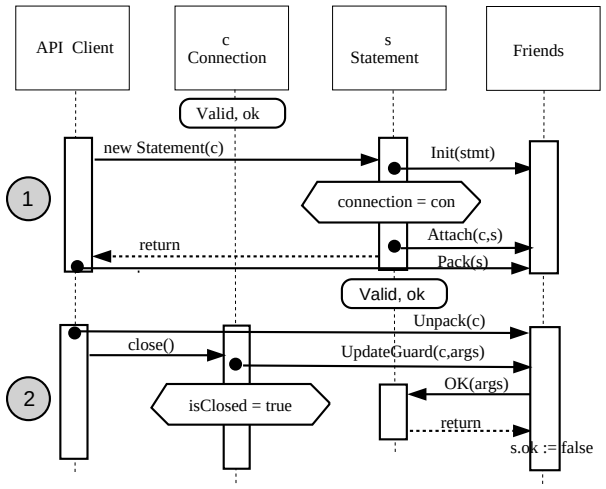

Figure 9. Using *Friends* Protocol.

6. Experience

We present our experiences in implementing the above methodology in a tool INVCOP++ and using it to enforce API protocols. INVCOP++ has two components:(1) the Verifier implements the protocol correctness check described in Section 4.1, and (2) the Enforcer implements the program conformance check described in Section 4.2.

6.1 Protocol Correctness

The Verifier implements the protocol correctness check we saw in Section 4.1. In particular, it verifies that the object protocol action $Init$ establishes the protocol invariant and

Protocol	# of proof obligations	Verified	Time (millisec)
Representation Containment	5	yes	70
PrivilegedReader	6	yes	70
Boogie	6	yes	70
IndirectReference v1	7	no	70
IndirectReference v2	7	yes	70
Friends	7	yes	70

Table 1. Protocol Verification

the other protocol actions maintain the protocol invariant. The Verifier generates proof obligations to be verified by an automated theorem prover. If a proof obligation is not valid, then the theorem prover returns a counterexample which can help the protocol designer in correcting mistakes in the protocol. Our implementation uses the Simplify automatic theorem prover [14].

Table 1 summarizes the results of protocol verification. For each protocol, the columns show the number of proof obligations, whether it was verified and the time taken in milliseconds. IndirectReference v1 is obtained by commenting out the assumptions to the Unpack action in Figure 7. Simplify generates a counterexample stating that the protocol invariant cannot be preserved since o becomes invalid and its owner can still be valid. After correcting this mistake, the protocol IndirectReference v2 as shown in Figure 7 is verified correct.

6.2 Program Conformance

The Enforcer implements the program conformance check we saw in Section 4.2. For a given protocol description and a binding to API classes, the Enforcer generates an aspect. If the API clients include this aspect in their build, then protocol violations are detected exactly where they occur in client programs. Our implementation takes as input a protocol description in PROLANG and a binding written using AspectJ [1] syntax. It then generates an aspect which can be compiled using the AspectJ compiler. We discuss an example binding below.

Suppose that the API designer of binary search tree requires every node n in the tree to own the key k it points to. First, the class `Node` and the interface `Key` needs to be bound to the role $ObjWInv$. This is specified as shown below.

```
declare parents: Node implements ObjWInv;
declare parents: Key implements ObjWInv;

//Inv method for Key
public boolean Key.Inv() {
  return true;
}
```

Then, after the field `Node.key` is set, the protocol method Own needs to be invoked. This is specified as shown below.

```
pointcut setKey(ObjWInv n, ObjWInv k) :
  set(private Key Node.key)
  && target(n)
  && args(k);

after(ObjWInv n, ObjWInv k) returning :
  setKey(n,k) {
  Own(n,k);
}
```

The pointcut (a specification of interesting points in the program) `setKey` captures setting the field `Node.key`. The target is the node n whose field is being set and the argument is the key k. After the field `Node.key` is set, the protocol method Own is invoked.

Given a protocol description and a binding such as the one above as inputs, the Enforcer generates an aspect by combining the inputs. Each protocol method is mapped to a method in the aspect. Assume statements in protocol actions are translated to assert statements. Based on the binding, the protocol methods are invoked and the execution proceeds only if the precondition of the protocol method is satisfied. At runtime, a singleton instance of the generated aspect is created in the virtual machine and it enforces the associated protocol.

The generated aspect computes dependencies of an object o dynamically by recording all the objects and fields referenced during the execution of $o.\texttt{Inv}()$ as in our earlier tool INVCOP [18]. Whenever a field f of object p is changed, for all objects o such that o depends on $p.f$, the $Validate$ method from the PROLANG description of the protocol is called.

Table 2 shows results from enforcing object protocols on two real world APIs, JDOM and MySQL JDBC, and several other programs that illustrate violations typical of realistic design scenarios involving multiple API objects. The first column gives the API name, and the column "Num Classes" gives the number of classes exposed by the API. The column "Scenario" gives the protocol scenario enforced. Table 3 gives more details on each scenario. The column "Protocol" gives the name of the object protocol on which the program is checked for conformance. The column "n" gives the number of scenarios executed and "d" gives the number of objects that each object depends upon. The column INVCOP shows the time taken to check protocol correctness and program conformance at runtime with our old tool INVCOP. The column INVCOP++ shows the time taken to

API	Num Classes	Scenario	Protocol	n	d	INVCOP (ms)	INVCOP++ (ms)
JDOM	69	S1	*Boogie*	2000	1	3915	3324
				4000	1	8452	5437
				8000	1	14461	9834
				16000	1	43663	21191
MySQL	95	S2	*Friends*	900	900	3055	160
				1000	1000	4046	161
				2000	2000	*	180
				4000	4000	*	231
BinarySearchTree	3	S3	*Representation Containment*	50	50	2043	1011
				100	100	14942	7721
				150	150	53788	22893
				200	200	124489	59596
BinarySearchTree	3	S4	*Representation Containment* variant	100	100	5618	220
				200	200	39707	540
				400	400	*	1802
				800	800	*	7360
Patient Observations	3	S5	*Representation Containment*	2000	1	471	280
				4000	1	831	490
				8000	1	1492	841
				16000	1	2714	1492
Deserialization	3	S6	*Representation Containment*	2000	1	140	50
				4000	1	170	50
				8000	1	250	80
				16000	1	371	110

Table 2. Execution time in milliseconds for INVCOP and INVCOP++, illustrating performance improvement obtained by combining static and dynamic analysis. The column labeled "n" shows the number of times a usage scenario was executed. The column labeled "d" shows the number of objects that depend on a protocol field and therefore the number of times $Validate$ is called when such a field changes. The value "Num Classes"shows the number of API classes. A cell with a "*" indicates that the time taken is more than 3 minutes.

Scenario	Num Classes involved	Description
S1	3	Iterate over an XML document and remove certain elements through the iterator
S2	3	Create statements using the same connection
S3	3	Search keys in a tree
S4	2	Insert keys into a tree
S5	2	Print observations through patient
S6	2	Access content of review through deserialized manager

Table 3. Description of scenarios

check program conformance with our new tool INVCOP++. When the number of dependents for an object is small, we get a 2X performance improvement using static analysis, When the number of dependents is large, the performance improvement is even larger, and for large values of d, the time taken by the unoptimized tool INVCOP is unacceptably high.

As described earlier, in the old tool INVCOP the object protocol is encoded into the program as described in the first part of Section 3 and therefore protocol correctness and program conformance are checked at runtime. The new tool INVCOP++ exploits the static analysis (i.e, the protocol correctness check) to optimize the runtime check. In particular, no validation checks are triggered for updates to protocol fields since the protocol has been statically validated. The only runtime checks are (1) the assertions that arise from the assume statements in the methods of the PROLANG description of the protocol, and (2) the dependency checks that arise

from updates to the existing fields of the objects (excluding the auxiliary fields used to model the protocol). Below, we give details on the APIs for each scenario in Table 2. For more details, see [27].

6.2.1 JDOM [23]

JDOM is an API that facilitates in-memory representation of XML documents and iterating over and manipulating the content of such documents. Figure 10 shows a usage of class `Document` in JDOM. 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 client code calls `detach` on an element, then it is removed from the list of nodes at that level. The iterator becomes invalid if the underlying collection is modified without its knowledge. This happens if the client code invokes `detach` during iteration. The iterator throws `ConcurrentModificationException` if the `next` method is called again. From the exception trace, it is difficult for JDOM developers to find out where exactly the API protocol was violated [24].

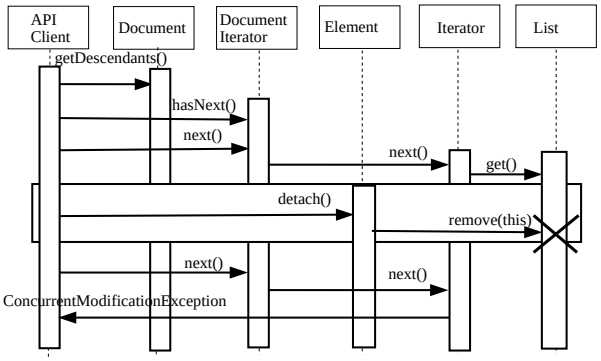

Figure 10. Navigating an XML Document.

After compiling with the aspect generated by InvCOP++ from the Boogie rule, the stack trace is as shown below. Here, the iterator owns the list when iteration starts and gives up ownership of the list after iteration ends.

```
java.lang.AssertionError:
Unpack(o)
o.st != State.Valid, o.st = Committed
o.class = class jdom.ListProxy
 at rules.Boogie_jdom.Unpack(Boogie_jdom.aj:319)
 ..
 at org.jdom.Content.detach(Content.java:91)
 at ItemHandler.processItem(OrderHandler.java:10)
 at OrderHandler.processOrder(OrderHandler.java:23)
```

This clearly points out that the client code `processItem` violated the API protocol by calling `Content.detach` which modifies the list without the knowledge of the iterator. Note that a class C can be bound to the role $ObjWInv$ by AspectJ only if the byte code of C is under its control. Since we cannot bind `java.util.Iterator` to $ObjWInv$, our prototype implementation uses proxy objects to keep track of the relationship between an iterator and its collection.

6.2.2 MySQL JDBC [28]

As we discussed in Section 5.3, a connection must not be closed before closing any statement created by that connection. However, JDBC API programmers make such mistakes [21] and it is hard to locate where exactly the connection was closed when a statement depending on that connection is still open.

The Friends protocol shown in Figure 8 is bound to classes `Statement` and `Connection` such that all the statements s created using a connection c share the connection c and c can be closed only if all statements s that depend on it are closed.

6.2.3 Patient and Observations [17]

Consider a scenario where two objects need to be merged. For example, in a hospital management system, on admitting a patient, a record is created. Later, it might be discovered that a separate patient record had already been created for the same patient. It is important to tie the two records together because the subsequent treatment might depend on the observations in both the records. For this, the API designer uses a superseding strategy in which one patient object is marked as active and the other object for the same patient as superseded. The superseded object is not deleted because it contains the information based on which a patient was treated before the objects were tied together. All data in the superseded object is copied to the active object and method calls on the superseded object are delegated to the active object.

The clients of this API may not know about the details of the superseding strategy and therefore may write a program in which a reference to the observations part of patient is retained assuming that it contains all the observations for a patient. Later, if two patient objects are merged, then this assumption does not hold any longer and could cause errors as shown in Figure 11.

The *Representation Containment* protocol can be used to prevent an API client from directly accessing observations without accessing patient. If the API designer wants clients to iterate over observations, then the *Privileged Reader* protocol can be used.

6.2.4 Deserialization [29]

A class may depend on the fact that the objects it refers to through private fields are not available through object references outside the class. For example, consider classes `Manager` and `Reviews`. Even if a manager refers to the reviews object through a private field, it may be possible to steal a reference to the reviews object during deserialization. Using such a reference, the reviews could be read without the

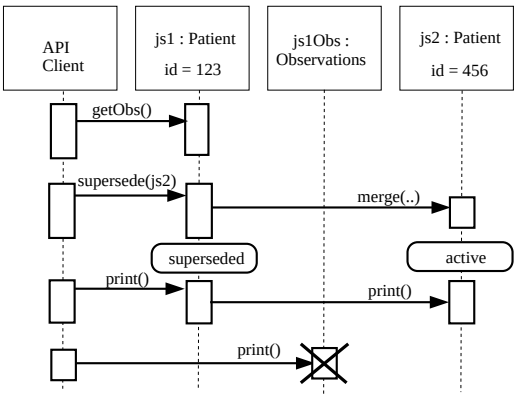

Figure 11. API Client makes wrong assumption.

knowledge of a manager. The *Representation Containment* protocol can be used to prevent this.

7. Related Work

Several papers have pointed out the need to specify and check protocols involving multiple objects. In [19], behavioral compositions and obligations on participants have been identified as key to object oriented design. Recently, [20] has pointed out the need to enforce framework constraints (which typically involve multiple objects) so that plugin writers cannot violate them.

Several *ownership* type systems have been invented to track dependencies between objects [9, 6]. 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 [3]. When multiple objects depend on a shared object (as in many statements depend on the same connection), the methodology needs to be extended [5]. Also, these systems do not work with existing programming languages. For each protocol, one has to use the corresponding verification system that works only for that protocol. We have been able to encode, verify and enforce all these protocols uniformly in our work. With our approach, the user can "mix-and-match" various protocols for various parts of the API, and "fine-tune" the protocol while still being able to use INVCOP++ to verify and enforce the protocol. However, the price paid for this flexibility is that our program conformance check is dynamic, whereas ownership type systems and program verification tools are able to check program conformance statically.

JML [26] requires that an invariant must hold at the end of each constructor's execution, and at the beginning and end of all public methods. JML's runtime checker does not automatically track dependencies of objects. Thus, if o depends on p, and p gets modified in a way that violates o's invariant, the JML checker is not able to detect this at the point of violation.

MOP [7] also generates aspects for runtime verification from specifications. It is possible that such aspects acting as observers will miss relevant events in the program. Therefore, one cannot guarantee that protocol violations are detected exactly where they occur in API client programs. With INVCOP++, we offer the guarantee as stated in Theorem 3.

8. Conclusion

We have presented a methodology to check if client programs that use object oriented APIs satisfy API protocol invariants.

Our methodology involves stating reusable object protocols involving inter-related objects in PROLANG. Our tool INVCOP++ checks statically that methods of a protocol establish and maintain its protocol invariant and checks at runtime whether a program conforms to a protocol. We have validated this methodology by stating and verify several object protocols in PROLANG and using INVCOP++ to detect protocol violations reported in discussion forums on widely used APIs. By judiciously combining static and dynamic analysis, we are able to reduce the performance overhead of runtime checking.

When compared to approaches such as ownership type systems, our approach is far more flexible, and allows "mix-and-match" and "fine-tuning" of object protocols. However, we are able to check program conformance only at runtime. Checking program conformance statically with an arbitrary protocol specified in PROLANG requires further research. Also, extending our work to concurrent programs, and handling subclasses, require further research.

References

[1] AspectJ – http://www.eclipse.org/aspectj/.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.

[3] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6):27–56, 2004.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 04: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, LNCS 3362. Springer Verlag, 2004.

[5] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84. Springer-Verlag, 2004.

[6] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223. ACM, 2003.

[7] F. Chen and G. Rosu. Mop: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, 2007.

[8] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI 05: Programming Language Design and Implementation*, pages 85–95. ACM, 2005.

[9] D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[10] `http://www.servlets.com/archive/servlet/ReadMsg?msgId=539019&listName=jdom-interest`.

[11] `http://bugs.mysql.com/bug.php?id=2054`.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1992.

[13] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation*. ACM, 2001.

[14] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[15] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA 06: Software Testing and Analysis*. ACM, 2006.

[16] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI 02: Programming Language Design and Implementation*, pages 1–12. ACM, 2002.

[17] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

[18] M. Gopinathan and S. Rajamani. Runtime monitoring of object invariants with guarantee. In *RV '08: Runtime Verification (to appear)*, 2008. available at `http://research.microsoft.com/~sriram/Papers/rv08.pdf`.

[19] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *OOPSLA/ECOOP*, pages 169–180, 1990.

[20] C. Jaspan and J. Aldrich. Checking framework plugins. In *OOPSLA Companion*, pages 795–796, 2007.

[21] `http://archives.postgresql.org/pgsql-jdbc/2003-10/msg00062.php`.

[22] `http://java.sun.com/products/jdbc/download.html#corespec40`.

[23] JDOM – `http://www.jdom.org`.

[24] JDOM FAQ – `http://www.jdom.org/docs/faq.html#a0390`.

[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[26] G. Leavens and Y. Cheon. Design by contract with jml, 2003.

[27] `http://people.csa.iisc.ernet.in/~gmadhu/oopsla`.

[28] MySQL – `http://www.mysql.com`.

[29] `http://java.sun.com/javase/6/docs/platform/serialization/spec/security.html#4271`.