Statically Checking API Protocol Conformance with Mined Multi-Object Specifications
— Companion Report —

Michael Pradel[1], Ciera Jaspan[2], Jonathan Aldrich[2], and Thomas R. Gross[1]

[1] Department of Computer Science, ETH Zurich
[2] Institute for Software Research, Carnegie Mellon University

Technical Report #752

ETH Zurich, Department of Computer Science

March 2012

Statically Checking API Protocol Conformance with Mined Multi-Object Specifications

— Companion Report —

Michael Pradel
Department of Computer Science
ETH Zurich, Switzerland

Ciera Jaspan Jonathan Aldrich
Institute for Software Research
Carnegie Mellon University

Thomas R. Gross
Department of Computer Science
ETH Zurich, Switzerland

***Abstract*—This technical report provides additional details for the paper entitled *Statically Checking API Protocol Conformance with Mined Multi-Object Specifications* [1]. We formally describe how to translate API usage protocols represented as finite state machines into a relationship-based specification language.**

I. Introduction

Programmers using an API often must follow protocols that specify when it is legal to call particular methods. Several techniques have been proposed to find violations of such protocols based on mined specifications. However, existing techniques either focus on single-object protocols or on particular kinds of bugs, such as missing method calls. There is no practical technique to find multi-object protocol bugs without a priori known specifications.

In [1], we combine a dynamic analysis that infers multi-object protocols and a static checker of API usage constraints into a fully automatic protocol conformance checker. The combined system statically detects illegal uses of an API without human-written specifications. Our approach finds 41 bugs and code smells in mature, real-world Java programs with a true positive rate of 51%. Furthermore, we show that the analysis reveals bugs not found by state of the art approaches.

This companion report provides a more detailed description of translating multi-object protocols into relationship-based specifications. Furthermore, we list all issues reported by the analysis to allow others to compare to our results.

II. Background

Our approach uses a dynamic specification miner and a static checker to respectively produce and check specifications of multi-object protocols. Both analyses divide a protocol into two parts. The first part determines the applicability of a protocol, and the second part describes the constraints imposed by the protocol.

A. Specification Mining and API Usage Protocols

We use a dynamic specification miner that extracts API usage protocols from training programs [2], [3]. Any existing API client can serve as a training program. A protocol consists of a deterministic finite state machine and a set of typed protocol parameters. States represent the common state of multiple objects or, for single-object protocols, the state of a single object. Transitions are labeled with method signatures that are annotated with protocol parameters naming the receiver, the method parameters and the return value.

Definition 1 (API usage protocol)**.** *An API usage protocol $\mathcal{P} = (M, P)$ consists of a deterministic finite state machine M and a finite set of typed protocol parameters P. M is a tuple $(S, \Sigma, \delta, s_0, S_f)$ of states S, the alphabet Σ, transitions δ, the initial state $s_0 \in S$, and final states $S_f \subseteq S$. A transition is a triple from $S \times \Sigma \times S$, which defines the source state, the label, and the destination state of the transition. The alphabet Σ consists of method signatures that are annotated with protocol parameters naming the receiver, and optionally, the method parameters and the return value.*

The mined protocols distinguish two kinds of states: *setup states* and *liable states*. The setup states establish which objects interact in the protocol by binding objects to protocol parameters. The liable states describe constraints on method calls that a programmer ought to respect. The miner constructs protocols in such a way that the set of parameters bound at a state is unambiguous. States at which all parameters are bound are liable states; all other states are setup states.

Figure 2 shows an API usage protocol describing the interplay of a collection c and an iterator i. The protocol specifies how to use the iterator (call `hasNext()` before `next()`) and that updating the collection invalidates the iterator. Calling the method `iterator()` on c, which returns i, establishes the interaction between these two objects. Therefore, the later states (3, 4, and 5) are all liable states. The table in Figure 2 gives the set of bound protocol parameters for each state.

Since we presented the protocol miner in [2], [3], it has evolved in several respects: First, we now create a single protocol per set of interacting types, so that, for example, a single protocol describes all constraints of the interplay between a collection and an iterator. Second, we use a heuristic to generalize protocols that contain overloaded methods. If calling an overloaded method is allowed at a particular state, then the protocol also allows calling other methods with the same name and the same number of parameters at this state. Third, we add a heuristic that allows calling pure methods at every state. We consider a method to be pure if the method does not change the state of any object and if the method does not throw any exception. Finally, we configure the protocol miner to ignore all calls to `Object` and `String`, because these calls occur very frequently in Java programs and dilute the protocols of other types.

B. Relationship-based Static Checking

This work uses Fusion [4], a relationship-based, static analysis, to check API clients against specifications over multiple objects. Fusion encodes usage constraints based on *relationships* between objects. A relationship is a user-defined, uninterpreted predicate across objects. For example, a binary relationship $r_{contains}(List, Element)$ can express that an object of type $List$ contains another object of type $Element$, such as $r_{contains}(list, element)$. If a relationship predicate is true, we say that the objects are in the relationship. Likewise, to say that the objects are not in a relationship means that the relationship predicate evaluates to false.

In Fusion, API methods are specified with *requirements* and *effects*.[1] A *requirement* is a logical proposition over relationship predicates that describes a precondition on a method. For example, `list.remove(element)` may require that $r_{contains}(list, element)$ holds. An *effect* is a postcondition that describes how the truth of relationship predicates changes after calling a method. For instance, `list.remove(element)` may have the effect to remove $(list, element)$ from $r_{contains}$. Both requirements and effects are guarded by a *trigger*, a logical proposition that describes when the requirement or effect applies.

Definition 2 (Effect constraint)**.** *An effect constraint on a method is a tuple (m, g, e), where m is the method to constrain, g is the trigger, and e is the set of changes to make to the state of the relationships.*

Definition 3 (Requirement constraint)**.** *A requirement constraint on a method is a tuple (m, g, q), where m is the method to constrain, g is the trigger, and q is the requirement for the method.*

A complete specification of a protocol in Fusion is a set of relationships, a set of effect constraints, and a set of requirement constraints on the relevant methods.

Definition 4 (Fusion specification of a protocol)**.** *A Fusion specification of a protocol can be described as $\mathcal{F} = (R, E, Q)$, where R is a set of relationships, E is a set of effect constraints, and Q is a set of requirement constraints.*

[1]We omit parts of Fusion not relevant for this work.

```
1 Collection c = ...
2 Iterator i = c.iterator();
3 if (i.hasNext())
4    System.out.println(i.next());
5 /* current state unknown */
6 c.update();        // legal -- invalidates iterator
7 if (i.hasNext())   // bug: iterator not valid anymore
8    System.out.println(i.next());
```

Figure 1: The state in line 5 cannot be determined because different paths lead to it.

Based on specifications of API methods, Fusion performs an intraprocedural analysis of API clients to check whether they respect the usage constraints. For each call to a method with a specification, the analysis checks whether all triggered requirements are fulfilled and applies all triggered effects. We use a complete variant of Fusion, which guarantees that any bug found will actually occur dynamically.

III. Translating Mined Protocols into Relationship-based Specifications

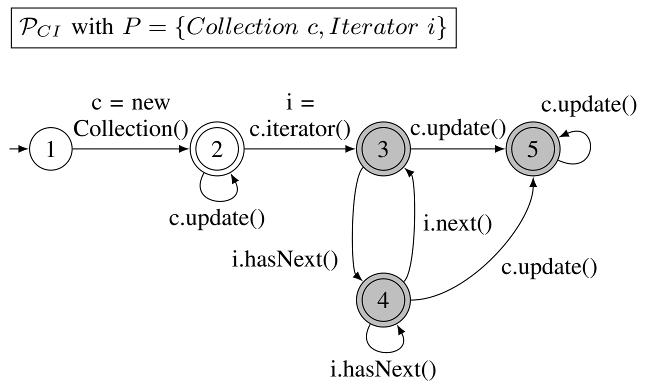
In the following, we describe how to combine protocol mining and relationship-based static checking by translating mined protocols into checkable specifications. We first discuss the main challenges for combining two formalisms like these in Section III-A. Then, Section III-B provides a high-level overview of the translation, followed by a more detailed description in Section III-C. Finally, Section III-D discusses how our approach addresses the challenges.

A. Challenges

We must deal with two main challenges, the first being common to all static analyses and the second being specific to multi-object checking:

1) Limited static knowledge: Static analysis inherently lacks precise knowledge about which program path is taken and as a result may miss bugs or report false positives. For example, Figure 1 shows a piece of source code where static analysis cannot determine the protocol state in line 5. After the call to `hasNext`, the protocol is known to be at state 4. The branch creates two paths, one path on which `next()` is called, leading back to state 3, and another path on which we stay at state 4. Because static analysis cannot determine which path is taken, the state in line 5 is unknown. Our combined analysis should nevertheless find the call in line 6 to be legal and detect the protocol violation in line 7.

2) Object interactions: Checking multi-object protocols is challenging, because calling a method on one object can influence the state of other objects. For example, the call to `update()` in line 6 does not refer to the iterator `i`, but directly affects its state. This dependence is implicit in mined



Bound protocol parameters:	s	1	2	3	4	5
	$bound(s)$	$\{\}$	$\{c\}$	$\{c,i\}$	$\{c,i\}$	$\{c,i\}$

Figure 2: Protocol describing how to use a collection and an iterator. A label $p = m()$ means that the object returned by $m()$ is bound to protocol parameter p. The call `c.update()` summarizes calls that may change the collection's content, for example, `c.add()` or `c.remove()`. Liable states have a gray background.

multi-object protocols, where a state is the common state of all involved objects.

B. Overview of the Translation

Given a set of protocols, how can we generate relationship-based specifications in a way that addresses the above challenges? We use relationships to reason about three aspects of objects with respect to a protocol:

1) We keep track of whether calling a method is allowed with a *method relationship*. This is a unary relationship for a method of the protocol. If an object is in a method relationship, it means that one can legally call the corresponding method on this object. The protocol in Figure 2 has four method relationships:

$$r_{next}(Iterator)$$
$$r_{hasNext}(Iterator)$$
$$r_{update}(Collection)$$
$$r_{iterator}(Collection)$$

There is no method relationship for constructor calls because a newly created object cannot be in any relationship before the constructor call.

2) We keep track of the current state of an object or a set of objects with *state relationships*. The semantics of a state relationship is that if an object or a set of objects is in this relationship, we know these objects to be at the corresponding state. The state relationship is over all objects that are bound at that state. The states of the protocol in Figure 2 translate into four state relationships:

$$r_2(Collection)$$
$$r_3(Collection, Iterator)$$
$$r_4(Collection, Iterator)$$
$$r_5(Collection, Iterator)$$

There is no state relationship for the initial state because no variables are bound at this state.

3) We keep track of which objects interact as described by a multi-object protocol, that is, which objects are "in the protocol". For this purpose, we create a *protocol relationship* over all protocol parameters. The semantics of the protocol relationship is that if a set of objects is in this relationship, we know these objects to interact with each other in the protocol. For example, we create a binary relationship $r_{\mathcal{P}_{CI}}(Collection, Iterator)$ for the two-object protocol $\mathcal{P}_{CI}$.

Protocol effects of calling `i = c.iterator()`	**Effects on relationships**
Move to state 3.	Set $r_3(c,i)$ to true; set $r_2(c)$, $r_4(c,i)$, and $r_5(c,i)$ to false.
Enable methods `i.hasNext()` and `c.update()`; disable methods `i.next()` and `c.iterator()`.	Set $r_{hasNext}(i)$ and $r_{update}(c)$ to true; set $r_{next}(i)$ and $r_{iterator}(c)$ to false.
Establish the interaction between `c` and `i`.	Set $r_{\mathcal{P}_{CI}}(c,i)$ to true.

Table I: Example of effects applied when calling `iterator()`.

The Fusion analysis uses relationships as data structures to describe the overall state of a protocol and to determine whether a method call is legal. Each method call in a program will be checked against the method's requirement constraints to ensure that the call is legal. Furthermore, each method call can change the state of the relationships according to the method's effect constraints. The overall approach is to use the method relationships to determine whether a method call is allowed. The protocol relationship and the state relationships are used to determine whether the protocol is at a liable state and which effects to make.

1) Making effects: Calling a method can have effects on the current state, on the currently enabled and disabled methods, and on the binding of objects to protocols. We create specifications that reflect these effects by adapting the corresponding relationships. For example, calling `i = c.iterator()` as described by the protocol in Figure 2 influences the state maintained by the analysis as shown in Table I.

Knowing that a particular method is called may not be sufficient to determine which effects to apply and which re-

Symbol	Meaning
$\mathcal{P}$	Protocol (M, P)
M	Finite state machine $(S, \Sigma, \delta, s_0, S_f)$
P, p	Set of protocol parameters, protocol parameter
S, s, t	Set of states, states
Σ, m	Alphabet, method signature $(m \in \Sigma)$
δ	Transitions $(\delta : S \times \Sigma \times S)$
$\mathcal{F}$	Fusion specification of a protocol (R, E, Q)
R, r	Set of relationships, relationship
E	Set of effect constraints (m, g, e)
Q	Set of requirement constraints (m, g, q)
g	Trigger (relationship predicate)
e	Effects (setting relationships to true and false)
q	Requirement (relationship predicate)

Table II: Reference of symbols used in this paper.

quirements to check. The reason is that a single method can label multiple transitions. For example, calling `c.update()` at state 2 (leading again to state 2) has different effects than calling the same method at state 3 (leading to state 5). We handle this problem by guarding effects with triggers on state relationships. In the example for `c.update()`, the translation creates two constraints in Fusion, one triggered by $r_2(c)$ and the other triggered by $r_3(c, i)$.

2) Checking requirements for method calls: To check whether calling a method is legal, we require that the receiver is in the corresponding method relationship. For example, to call `i.next()`, we require that $r_{next}(i)$ is true.

The requirements imposed by a protocol are only valid if the current state is a liable state. Therefore, we guard requirements with the protocol relationship, ensuring that the involved objects interact in the protocol and that the protocol is in a liable state. For example, the requirement on calling `i.next()` is guarded by $r_{\mathcal{P}_{CI}}(c, i)$. This can be thought of as a logical precondition of the form:

$$\forall c \;.\; (r_{\mathcal{P}_{CI}}(c, i) \implies r_{next}(i))$$

C. Formalization

This section provides a more detailed, formal description of how to translate a finite state machine representation of a multi-object protocol into a logical predicate representation suitable for Fusion. Table II lists all symbols we use in this section along with their meaning.

The translation uses two functions:

Definition 5 ($Enabled : S \to 2^{\Sigma}$)**.** *The $Enabled(s)$ function returns all the methods that are legal to call at state s. That is, $Enabled(s) = \{m \mid m \in \Sigma \;\wedge\; \exists t \;.\; (s, m, t) \in \delta\}$.*

Definition 6 ($Disabled : S \to 2^{\Sigma}$)**.** *The $Disabled(s)$ function returns all the methods that are illegal to call at state s. That is, $Disabled(s) = \{m \mid m \in \Sigma \;\wedge\; \nexists t \;.\; (s, m, t) \in \delta\}$.*

We use the following abbreviated notation for relationships by making the objects implicit:

- A protocol relationship $r_{\mathcal{P}}$ means $r_{\mathcal{P}}(p_1, .., p_{|P|})$, where $p_1, .., p_{|P|}$ are the protocol parameters P of protocol $\mathcal{P}$.
- A state relationship r_s means $r_s(p_1, .., p_{|bound(s)|})$, where $p_1, .., p_{|bound(s)|}$ are the protocol parameters $bound(s)$.
- A method relationship r_m means $r_m(p)$, where p is the receiver of m.
- We denote the effect to set a relationship to true as $+r$ and the effect to set a relationship to false as $-r$.

Having defined the above helper functions and our notation, we can now give a formal, declarative description of how to translate a protocol $\mathcal{P} = (M, P)$ into a Fusion specification $\mathcal{F} = (R, E, Q)$. The translation consists of three steps. First, the translation creates relationships R that represent states, methods, and the protocol itself, as described in Section III-B. Second, the translation creates a set of effect constraints E for each method occurring in the protocol. These constraints control how calling a method influences the state of relationships. Third, the translation creates a set of requirement constraints Q, which specify preconditions for calling methods. The following describes creating effect constraints and requirement constraints, and illustrates the translation on the protocol in Figure 2.

1) Effect constraints with known state.: The translation creates effect constraints that represent the effects that occur when taking transitions. For each transition $(s, m, t) \in \delta$, there will be an effect constraint $(m, g, e) \in E$ where

$$g = \begin{cases} true & \text{if } s = s_0 \\ r_s \wedge r_{\mathcal{P}} & \text{if } s \text{ is a liable state} \\ r_s & \text{if } s \text{ is a setup state} \end{cases}$$

$$e = +r_t \;\cup \Big(\bigcup_{s' \in S \setminus \{t\}} -r_{s'} \Big) \;\cup\; \Big(\bigcup_{m' \in Enabled(t)} +r_{m'} \Big) \;\cup \Big(\bigcup_{m' \in Disabled(t)} -r_{m'} \Big) \;\cup\; e_{\mathcal{P}}$$

$$\text{where } e_{\mathcal{P}} = \begin{cases} +r_{\mathcal{P}} & \text{if } m \text{ combines the protocol parameters} \\ \text{none} & \text{otherwise} \end{cases}$$

In the example, there are four transitions labeled with `c.update()`, which produce the four effect constraints labeled with $\diamond$ in Figure 3a.

Notice that e contains three kinds of effects. First, we set the state relationship r_t of the destination state t to true and set state relationships of all other states to false. Second, we enable all methods that can be legally called at the target state t and disable all other methods. Finally, if m combines protocol parameters, we set the protocol relationship to true for these objects to establish that these objects interact as described by the protocol.

The effect constraints are triggered by g in one of three ways. If s is the initial state, the trigger must be true since there is no prior knowledge for this protocol. If s is a setup state, the trigger is the state relationship r_s. Finally, if s is a liable state, the trigger is $r_s \ \wedge \ r_{\mathcal{P}}$ to ensure that the effect only occurs when the current state of the protocol is liable.

2) Effect constraint with unknown state.: The Fusion analysis is a dataflow analysis and can lose precision after merging paths with different state information. Therefore, the translation must also handle the case where a method call within a protocol occurs while we have no information about the current state. To deal with this case, the translation makes an effect constraint for each method. This constraint considers all possible states at which we can be based on the available knowledge.

Let $S_t(m)$ be the set of target states of m, where $S_t(m) = \{t \mid \exists s \,.\, (s, m, t) \in \delta \wedge s \text{ is a liable state}\}$. For each $m \in \Sigma$, there will be an effect constraint $(m, g, e) \in E$ where

$$\begin{aligned} g &= r_{\mathcal{P}} \\ e &= e_{state} \cup (\bigcup_{m' \in AlwaysEnabled} +r_{m'}) \\ &\quad \cup (\bigcup_{m' \in AlwaysDisabled} -r_{m'}) \end{aligned}$$

where

$$\begin{aligned} e_{state} &= \begin{cases} r_t & \text{if } S_t = \{t\} \\ \text{none} & \text{otherwise} \end{cases} \\ AlwaysEnabled &= \bigcap_{t' \in S_t} Enabled(t') \\ AlwaysDisabled &= \bigcap_{t' \in S_t} Disabled(t') \end{aligned}$$

In the example, `c.update()` produces the effect constraint labeled with $\star$ in Figure 3a.

To ensure that the involved objects are indeed interacting in the protocol whenever the effect applies, the translation guards the effect with a trigger on the protocol relationship. However, since the current state is unknown, the translation applies all effects of calling m that are independent of which transition labeled with m is taken. In the trivial case, where m labels a single transition in $\mathcal{P}$, the effects are the same as the state-dependent effects described in Section III-C1.

Triggers of effect constraints have two purposes. First, triggers guard effects to apply them only if the effects are applicable. For example, we check state relationships to distinguish between the same method call from different states. Second, triggers make objects visible in the scope of a constraint. For example, if a transition leads from a state where two objects are bound to another state where two objects are bound, we want to update the state relationship for both objects. However, the call may expose only one object (the receiver), leaving the other object invisible. This is the case for calls to `c.update()`, where we must update the relationships on both the collection and any associated iterators. By guarding the effect with a trigger over $r_{\mathcal{P}_{CI}}(c, i)$, the iterator becomes bound and the analysis can apply effects on all necessary relationships.

All effect constraints for the example are summarized in Figure 3a. Each line of the table shows one effect constraint with its method, its trigger, and its effects. While some generated constraints are logically included in others and can be omitted without influencing the results of the static analysis, this is not true in general, so the translation generates all effects as described here.

Requirement constraints.: The requirement constraints that control when a method can be called are very simple: For each $m \in \Sigma$, the translation creates a single requirement constraint (m, g, q) in Q, where $g = r_{\mathcal{P}}$ and $q = r_m$. That is, we specify that for calling m, the receiver of m must be in m's method relationship. Figure 3b lists the requirement constraints we generate for the example.

D. Meeting the Challenges

Our revised approach meets the challenges mentioned in Section III-A. We deal with limited static knowledge by maintaining both the current state and the currently enabled and disabled methods. This approach allows the analysis to recover knowledge that has been lost when merging paths. Even when the current state is unknown, the analysis may still know about currently enabled methods and continue the analysis based on this information. The translation addresses the problem of interacting objects by representing states at which n objects are bound by n-ary state relationships. Furthermore, we represent the interaction of objects through the protocol relationship. By using the protocol relationship as a trigger, the analysis can apply effects on all involved objects, even if a method references only a subset of them.

Translating the protocol in Figure 2 gives constraints that detect the bug in Figure 1. Before checking line 6, the analysis knows the relationships $r_{update}(c)$ and $r_{prot}(c, i)$. That is, calling `c.update()` is legal and c is known to interact with i. The effects of the call lead to the following relationships after line 6: $r_5(c)$, $r_5(i)$, and $\neg r_{hasNext}(i)$. Thus, the precondition for calling `i.hasNext()` in line 7 does not hold and Fusion reports a warning.

IV. Reported Warnings

Tables III, IV, and V list all bugs, code smells, and false positives that the analysis reports.

Method m		Trigger g	Effects e		
			States	Methods	Protocol
`new Collection()`		true	$+r_2, -r_3, -r_4, -r_5$	$+r_{up}, +r_{it}, -r_{ha}, -r_{ne}$	—
`update()`	$\diamond$	r_2	$+r_2, -r_3, -r_4, -r_5$	$+r_{up}, +r_{it}, -r_{ha}, -r_{ne}$	—
`update()`	$\diamond$	$r_3 \wedge r_{\mathcal{P}_{CI}}$	$-r_2, -r_3, -r_4, +r_5$	$+r_{up}, -r_{it}, -r_{ha}, -r_{ne}$	—
`update()`	$\diamond$	$r_4 \wedge r_{\mathcal{P}_{CI}}$	$-r_2, -r_3, -r_4, +r_5$	$+r_{up}, -r_{it}, -r_{ha}, -r_{ne}$	—
`update()`	$\diamond$	$r_5 \wedge r_{\mathcal{P}_{CI}}$	$-r_2, -r_3, -r_4, +r_5$	$+r_{up}, -r_{it}, -r_{ha}, -r_{ne}$	—
`update()`	$\star$	$r_{\mathcal{P}_{CI}}$	$-r_2, -r_3, -r_4, +r_5$	$+r_{up}, -r_{it}, -r_{ha}, -r_{ne}$	—
`iterator()`		r_2	$-r_2, +r_3, -r_4, -r_5$	$+r_{up}, -r_{it}, +r_{ha}, -r_{ne}$	$+r_{\mathcal{P}_{CI}}$
`hasNext()`		$r_3 \wedge r_{\mathcal{P}_{CI}}$	$-r_2, -r_3, +r_4, -r_5$	$+r_{up}, -r_{it}, +r_{ha}, +r_{ne}$	—
`hasNext()`		$r_4 \wedge r_{\mathcal{P}_{CI}}$	$-r_2, -r_3, +r_4, -r_5$	$+r_{up}, -r_{it}, +r_{ha}, +r_{ne}$	—
`hasNext()`		$r_{\mathcal{P}_{CI}}$	$-r_2, -r_3, +r_4, -r_5$	$+r_{up}, -r_{it}, +r_{ha}, +r_{ne}$	—
`next()`		$r_4 \wedge r_{\mathcal{P}_{CI}}$	$-r_2, +r_3, -r_4, -r_5$	$+r_{up}, -r_{it}, +r_{ha}, -r_{ne}$	—
`next()`		$r_{\mathcal{P}_{CI}}$	$-r_2, +r_3, -r_4, -r_5$	$+r_{up}, -r_{it}, +r_{ha}, -r_{ne}$	—

(a) Effect constraints.

Method m	Trigger g	Reqmt. q
`update()`	$r_{\mathcal{P}_{CI}}$	r_{up}
`iterator()`	$r_{\mathcal{P}_{CI}}$	r_{it}
`hasNext()`	$r_{\mathcal{P}_{CI}}$	r_{ha}
`next()`	$r_{\mathcal{P}_{CI}}$	r_{ne}

(b) Requirement constraints.

Figure 3: Constraints generated for the protocol in Figure 2. Method relationships are abbreviated (r_{ne} means r_{next}, etc.).

Program	File	Line	Characters	Comment
avrora	avrora/sim/mcu/ATMegaTimer.java	331	10897–10905	illegal call to next(); should crash on every execution
avrora	avrora/sim/mcu/ATMegaTimer.java	331	10934–10942	illegal call to next(); should crash on every execution
avrora	avrora/sim/platform/PinConnect.java	319	12097–12124	concurrent modification
avrora	avrora/sim/platform/PinConnect.java	334	12627–12654	concurrent modification
avrora	avrora/sim/platform/PinConnect.java	447	16049–16057	illegal call to next(); pinConnections list is empty after initializing PinConnect, removing lines 447 and 448 preserves the functionality while making the iterator usage safe
avrora	cck/stat/Sequence.java	54	2258–2270	illegal call to next(); list fragments is initialized with one element in the constructor of Sequence; however, fragments is not private and may be cleaned by any other class in the package
avrora	jintgen/gen/CodeSimplifier.java	183	7581–7590	illegal call to next(); params and args are assumed to have the same length; will crash otherwise; e.g. called by visit(CallExpression, CGEnv), which may receive a CallExpression not fulfilling the assumption
avrora	jintgen/gen/disassembler/DisassemblerTestGenerator.java	64	2467–2489	illegal call to next(); list empty after initializing class SimpleValues
avrora	jintgen/gen/disassembler/DisassemblerTestGenerator.java	80	3062–3086	illegal call to next(); values are empty after initializing class CompoundValues
eclipse	org/eclipse/jdt/internal/eval/Evaluator.java	76	3446–3460	illegal call to next(); values() may return a set that is smaller than size()
eclipse	org/eclipse/update/internal/configurator/SiteEntry.java	419	13129–13144	illegal call to next(); values() may return a set that is smaller than size()
fop	org/apache/fop/fo/flow/table/CollapsingBorderResolver.java	214	8874–8887	illegal call to next(); depends on size of row, which is not specified/checked to be positive
fop	org/apache/fop/fo/flow/table/CollapsingBorderResolver.java	216	8995–9009	illegal call to next(); Table.getColumns() can return empty list: Table.columns is empty when creating a table (similar to other bug in this method)
fop	org/apache/fop/fo/flow/table/CollapsingBorderResolver.java	221	9330–9344	illegal call to next(); Table.getColumns() can return empty list: Table.columns is empty when creating a table
fop	org/apache/fop/layoutmgr/BlockStackingLayoutManager.java	592	25835–25857	illegal call to next(); crashes if unexpected 'oldList' is passed
fop	org/apache/fop/layoutmgr/ElementListUtils.java	75	3256–3267	illegal call to next(); parameter 'elements' can have less elements than expected, javadoc says nothing about minimum size
fop	org/apache/fop/layoutmgr/inline/TextLayoutManager.java	802	34624–34646	illegal call to next(); parameter 'oldList' can have less elements than expected, javadoc says nothing about minimum size
fop	org/apache/fop/layoutmgr/inline/TextLayoutManager.java	803	34705–34727	illegal call to next(); parameter 'oldList' can have less elements than expected, javadoc says nothing about minimum size
fop	org/apache/fop/layoutmgr/inline/TextLayoutManager.java	832	36071–36093	illegal call to next(); parameter 'oldList' can have less elements than expected, javadoc says nothing about minimum size
jython	tests/java/javatests/ListTest.java	445	17972–18007	illegal call to next(); a bug that is intended to be a bug by the programmers (in a test case, to check whether an exception is thrown)
jython	tests/java/javatests/ListTest.java	493	20406–20417	a bug that is intended to be a bug by the programmers (in a test case, to check whether an exception is thrown)
lucene	org/apache/lucene/index/TestIndexReader.java	1321	52108–52126	the second call to iterator should be on fields2
lucene	org/apache/lucene/search/spell/TestLuceneDictionary.java	162	5670–5679	illegal call to next(); custom iterator doesn't follow the protocol of iterators specified in Iterator's javadoc (but returns null if there's no more element)
lucene	org/apache/lucene/search/spell/TestLuceneDictionary.java	163	5745–5754	illegal call to next(); duplicate of bug one line above (incorrect custom iterator)
pmd	regress/test/net/sourceforge/pmd/cpd/MatchAlgorithmTest.java	94	3613–3625	illegal call to next(); as far as I can see, Match.markSet typically has two elements, but never three; so this should fail when running
pmd	regress/test/net/sourceforge/pmd/cpd/MatchAlgorithmTest.java	95	3635–3647	illegal call to next(); Match.markSet can have size 1 if the TokenEntries passed to Match's constructor are the same (even if the size is 2, it's illegal here)

Table III: Bugs found by the analysis.

Program	File	Line	Characters	Comment
eclipse	org/eclipse/jface/text/templates/TemplateContextType.java	291	9582–9597	safety depends on the fact that positions.size $\geq$ variable.getOffsets.size, which is not documented anywhere (but which seems to be enforced by the only callee of this method)
fop	org/apache/fop/layoutmgr/ElementListUtils.java	70	3053–3064	code is based upon the assumption that 'elements' has a certain size and certain kinds of elements (which is not stated in the javadoc); however, the calls to previous() make this call kind of safe (the program may crash already at the call to previous())
jython	tests/java/javatests/ListTest.java	516	21447–21456	test depends on length of 'defaultList', which is fixed in the defaultList() method; nice example of hidden dependencies between methods; not a bug because it's a test
lucene	org/apache/lucene/analysis/de/TestGermanStemFilter.java	60	2275–2286	unnecessary call to close, closing the outer reader is enough
lucene	org/apache/lucene/search/spell/TestLuceneDictionary.java	152	5123–5132	safety depends on number of words in 'ld', which is set somewhere outside this method; not a bug, because it's a test
lucene	org/apache/lucene/search/spell/TestLuceneDictionary.java	153	5204–5213	safety depends on number of words in 'ld', which is set somewhere outside this method; not a bug, because it's a test
pmd	regress/test/net/sourceforge/pmd/cpd/MatchAlgorithmTest.java	65	2419–2431	Match.markSet can have size 1 if two equal objects are passed to Match's constructor
pmd	regress/test/net/sourceforge/pmd/jaxen/DocumentNavigatorTest.java	116	3866–3877	depends on size of collection, which is set somewhere else; not counted as bug because it's in a test, which should not be reused by anyone
pmd	regress/test/net/sourceforge/pmd/jaxen/DocumentNavigatorTest.java	118	3977–3988	depends on size of collection, which is set somewhere else; not classified as a bug because it's a test case that is probably not used by anyone else
pmd	regress/test/net/sourceforge/pmd/jaxen/DocumentNavigatorTest.java	90	2835–2846	depends on size of collection, which is set somewhere else; not counted as bug because it's in a test, which should not be reused by anyone
pmd	regress/test/net/sourceforge/pmd/RuleSetFactoryTest.java	673	29600–29631	is OK as long as all XML strings defined above contains at least one rule
pmd	regress/test/net/sourceforge/pmd/symboltable/ClassScopeTest.java	101	3969–3977	depends on size of collection, which is set somewhere else; not classified as a bug because it's a test case that is probably not used by anyone else
pmd	regress/test/net/sourceforge/pmd/symboltable/ClassScopeTest.java	218	9022–9030	depends on size of collection, which is set somewhere else; not classified as bug because it's in a test case
pmd	net/sourceforge/pmd/util/ClasspathClassLoader.java	72	2403–2413	duplicate close, which is not necessary (first close in try-block can be removed without having any effect)
xalan	org/apache/xml/utils/Hashtree2Node.java	129	4892–4901	correct, but maybe difficult to maintain; safe iteration depends on adding pairs of objects to 'v' (which happens in the same method)

Table IV: Code smells found by the analysis.

Program	File	Line	Characters	Comment
avrora	avrora/sim/radio/Medium.java	547	23345–23353	incomplete protocol; size() before next() is not part of the protocol
avrora	jintgen/gen/Inliner.java	137	4862–4877	incomplete protocol; assumes that d.params and args have the same size, checks for it at beginning of method, so it's okay
avrora	jintgen/isdl/verifier/TypeChecker.java	177	7096–7132	incomplete protocol; checks for the size of accessors before calling next()
avrora	jintgen/isdl/verifier/TypeChecker.java	189	7650–7662	incomplete protocol; checks that sizes of args and d.params() are the same
batik	org/apache/batik/apps/svgbrowser/Main.java	494	18310–18319	incomplete protocol
eclipse	core/framework/org/eclipse/osgi/framework/debug/DebugOptions.java	143	5056–5108	incomplete protocol
eclipse	dom/org/eclipse/jdt/core/dom/ASTMatcher.java	103	3880–3890	incomplete protocol
eclipse	model/org/eclipse/jdt/internal/core/DeltaProcessor.java	1417	53669–53693	incomplete protocol
eclipse	src-model/org/eclipse/core/internal/model/RegistryResolver.java	403	15309–15330	incomplete protocol
eclipse	src-model/org/eclipse/core/internal/model/RegistryResolver.java	421	15848–15871	incomplete protocol
eclipse	src-model/org/eclipse/core/internal/model/RegistryResolver.java	517	19162–19176	incomplete protocol
eclipse	org/eclipse/ant/core/AntCorePreferences.java	425	15297–15326	incomplete protocol
eclipse	org/eclipse/core/internal/indexing/PageStore.java	523	14973–14999	incomplete protocol
eclipse	org/eclipse/core/internal/refresh/RefreshJob.java	61	2206–2229	incomplete protocol
eclipse	org/eclipse/core/internal/watson/ElementTree.java	400	15085–15122	incomplete protocol
eclipse	org/eclipse/jface/text/TextUtilities.java	184	5906–5921	incomplete protocol
eclipse	org/eclipse/team/internal/core/subscribers/SyncInfoTreeChangeEvent.java	51	2010–2035	incomplete protocol
fop	org/apache/fop/fo/XMLWhiteSpaceHandler.java	257	10563–10580	incomplete protocol; see similar warning in this method
fop	org/apache/fop/fo/XMLWhiteSpaceHandler.java	284	11914–11931	incomplete protocol; see similar warning in this method
fop	org/apache/fop/fo/XMLWhiteSpaceHandler.java	299	12528–12545	incomplete protocol; interesting case, call to next seems to be missing; custom iterator provides nextChar which internally calls next
fop	org/apache/fop/layoutmgr/table/TableStepper.java	488	18966–19001	incomplete protocol
h2	org/h2/util/ObjectArray.java	35	1008–1017	incomplete protocol
jython	org/python/core/__builtin__.java	410	16877–16886	incomplete protocol
jython	org/python/core/PyUnicode.java	800	25063–25074	custom iterator, see related warning in same class
jython	org/python/core/PyUnicode.java	802	25130–25141	interesting case, custom iterator that offers another method (peek()) to check if there's a next element
jython	tests/java/org/python/expose/generate/ExposedTypeProcessorTest.java	32	1279–1324	incomplete protocol
lucene	org/apache/lucene/analysis/shingle/ShingleFilter.java	266	7941–7958	incomplete protocol
lucene	org/apache/lucene/AnalysisTest.java	57	1936–1946	mixture of incomplete protocol and limitation to intra-procedural analysis
lucene	org/apache/lucene/benchmark/byTask/tasks/RepAllTask.java	58	1952–1972	Fusion imprecision
lucene	org/apache/lucene/benchmark/byTask/tasks/RepSelectByPrefTask.java	56	1996–2016	Fusion imprecision
lucene	org/apache/lucene/index/TestIndexReader.java	1323	52235–52245	incomplete protocol
lucene	org/apache/lucene/index/TestIndexReader.java	1355	53515–53525	incomplete protocol; calls to size - so it's OK
lucene	org/apache/lucene/queryParser/QueryParser.java	1679	52986–53016	incomplete protocol; seems like a concurrent modification, but isn't because of the break
pmd	regress/test/net/sourceforge/pmd/cpd/MatchTest.java	30	1019–1027	size is (indirectly) set by creating the two tokens, i.e. in this method - so it's OK
pmd	regress/test/net/sourceforge/pmd/RuleSetFactoryTest.java	110	3393–3424	incomplete protocol; next is OK because size is checked before
pmd	regress/test/net/sourceforge/pmd/RuleSetTest.java	103	3243–3251	incomplete protocol
pmd	regress/test/net/sourceforge/pmd/util/ApplierTest.java	33	839–851	
pmd	net/sourceforge/pmd/jsp/ast/JspParser.java	1553	46004–46034	interesting case; seems like a concurrent modification but is correct because of the break; generated code
tomcat	org/apache/el/parser/ELParser.java	2100	59198–59228	no concurrent modification because of the break
tomcat	org/apache/jasper/compiler/TagFileProcessor.java	721	29586–29606	no concurrent modification here, because method returns immediatly

Table V: False positives reported by the analysis.

References

[1] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, “Statically checking API protocol conformance with mined multi-object specifications,” in *ICSE*, 2012.

[2] M. Pradel, P. Bichsel, and T. R. Gross, “A framework for the evaluation of specification miners based on finite state machines,” in *ICSM*, 2010, pp. 1–10.

[3] M. Pradel and T. R. Gross, “Automatic generation of object usage specifications from large method traces,” in *ASE*, 2009, pp. 371–382.

[4] C. Jaspan and J. Aldrich, “Checking framework interactions with relationships,” in *ECOOP*, 2009, pp. 27–51.