2009 IEEE/ACM International Conference on Automated Software Engineering

Mining Temporal Specifications from Object Usage

Andrzej Wasylkowski
Dept. of Computer Science
Saarland University, Saarbrücken, Germany
Email: wasylkowski@cs.uni-saarland.de

Andreas Zeller
Dept. of Computer Science
Saarland University, Saarbrücken, Germany
Email: zeller@cs.uni-saarland.de

***Abstract*—A caller must satisfy the callee's precondition—that is, reach a state in which the callee may be called. Preconditions describe the state that needs to be reached, but not how to reach it. We combine static analysis with model checking to mine *Computation Tree Logic* (CTL) formulas that describe the operations a parameter goes through: "In `parseProperties(String xml)`, the parameter `xml` normally stems from `getProperties()`." Such *operational preconditions* can be learned from program code, and the code can be checked for their violations. Applied to AspectJ, our Tikanga prototype found 189 violations of operational preconditions, uncovering 9 unique defects and 36 unique code smells—with 44% true positives in the 50 top-ranked violations.**

I. Introduction

When using a function, a client must ensure that the function's *precondition* is satisfied—the condition that has to be met before its execution. Even simple functions can have surprisingly complex preconditions. The AspectJ method `reapPropertyList`, for instance, takes a list as a parameter. In order to function properly, though, this list must be nonempty, its first element must be a class, and subsequent elements must be structural property descriptors, whose node class is equal to the class being the first element of the list. In a JML specification, this precondition reads as follows:

```
static List ASTNode.reapPropertyList(List list)
@requires list.size() >= 1
@requires list.get(0) instanceof Class
@requires \forall int i; 0 < i && i < list.size();
  list.get(i) instanceof StructuralPropertyDescriptor
  && ((StructuralPropertyDescriptor)list.get(i)).
     getNodeClass() == list.get(0)
```

This is an example of an *axiomatic* precondition, which is the base of several verification and validation approaches. While it describes the precise state of the program and the method's parameters, it does not tell *how to achieve this state*: Where does `list` come from? How do we construct it to satisfy the precondition?

To answer such questions, programmers usually refer to *usage examples* in existing code. In the case of `reapPropertyList`, we can examine one of its callers, say, `getPropertyList`, to learn that the `list` in question would be constructed from a set of properties. Without explicitly stating the `reapPropertyList` precondition, the code in Fig. 1 shows *how* to meet it.

Looking for such examples and extracting what *needs* to be done for the precondition to be satisfied is difficult and error-prone. We therefore introduce the concept of *operational preconditions* specifying *how* to satisfy a function's requirements. Operational preconditions come in the form of sets of *Computation Tree Logic (*CTL*) formulas* [8], expressing the control and data flow through function calls. The operational precondition for the `list` parameter of `reapPropertyList` would contain CTL formulas that describe conditions such as "The `list` is always eventually passed as a second argument to `createPropertyList`", or "After being passed to `createPropertyList`, the `list` may be passed to `addProperty`".

```
public List getPropertyList (Set properties) {
    List list = new ArrayList ();
    createPropertyList (this.cl, list);
    Iterator iter = properties.iterator ();
    while (iter.hasNext ()) {
        Property p = (Property) iter.next ();
        addProperty (p, list);
    }
    reapPropertyList (list);
    if (list.size () == 1)
        Debug.log ("Empty property list");
    return list;
}
```

Fig. 1. Sample usage of `reapPropertyList`.

Creating such specifications for project-specific classes is a lot of work that cannot be reused in other projects, and this is the problem we are addressing with Tikanga. Rules such as the ones above can be *mined* from existing source code, like the one shown in Fig. 1. The basic idea is as follows: *If a method is called sufficiently often, we can deduce its operational preconditions from its callers.* The preconditions may then become part of the documentation, for instance. For us, though, their most important use is to check for *anomalies*—that is, code fragments which violate the operational preconditions. Such anomalies uncover errors and code smells in the program code—simply because a method is being used in an uncommon way.

In this paper, we introduce Tikanga[1]—a tool that automatically learns and checks operational preconditions using a combination of static program analysis, model checking, and concept analysis. The process consists of four steps, each detailed in its own section (see Fig. 2 for an overview):

[1] "Tikanga" is the Māori word for "correct procedure".

1527-1366/09 $29.00 © 2009 IEEE
DOI 10.1109/ASE.2009.30

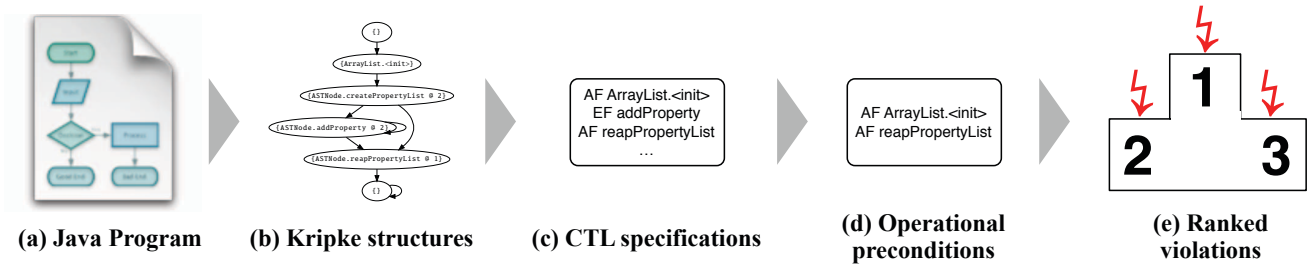

Fig. 2. TIKANGA in a nutshell. TIKANGA takes a JAVA program as input (a) and derives Kripke structures for the usages of methods' parameters (b). These are then used to extract CTL specifications (c). Multiple CTL specifications are generalized into operational preconditions (d). Violations of these preconditions are ranked and presented to the user (e) as potential defects.

Step 1: Model mining. TIKANGA takes a JAVA program as its input. It uses JADET [29] to create a so-called *object usage model* for each statically identifiable object in that program. It then transforms each object usage model produced by JADET into a set of *prefix models*, each focusing on a different *call event* that is part of the model. These prefix models are then transformed into *Kripke structures*, where transitions of the prefix model become labels (Section II).

Step 2: Extracting CTL *specifications.* For each Kripke structure, TIKANGA generates a set of CTL formulas stemming from user-given templates. Labels from the Kripke structure become atomic propositions in the formulas. These formulas are then model checked against the structure to determine which ones hold. They describe the way a specific object is used before being passed to a specific method as an argument (Section III).

To our knowledge, TIKANGA *is the first tool to mine quantified temporal specifications from code. Our approach is generic and easily* extensible to arbitrary CTL formulas.

Step 3: Creating operational preconditions. For each formal parameter of each method, TIKANGA looks for sets of CTL formulas that are *common to many objects* passed as actual arguments. Each such set forms an *operational precondition* of the formal parameter (Section IV).

We introduce the concept of operational preconditions, along with the tools and techniques to obtain and leverage them.

Step 4: Detecting violations. TIKANGA looks through all the objects used as actual arguments and identifies those that *violate* the operational preconditions found. These violations are reported to the user for investigation; violated CTL formulas suggest a potential fix (Section V).

TIKANGA *is the first tool to detect* anomalies *related to the usage of objects across several methods. It requires no specification or execution, and delivers results within minutes.*

We have evaluated TIKANGA on a set of six real-life JAVA programs (Section VI). The results are very promising: TIKANGA discovered previously unknown defects and code smells, many of them subtle, and all spread across multiple method invocations. Our tool also has a high true positive rate: In AspectJ, 44% of the 50 top-ranked anomalies were code issues.

II. CREATING MODELS

A. Object usage models

The first step in learning operational preconditions and detecting their violations is creating so-called *object usage models* for all statically identifiable objects in the program. These objects are: *formal parameters* of methods (including the implicit `this` parameter), *objects created* via `new`, *return values* of method calls (as in `x = map.items()`), values read from *fields* (including static fields, as in `x = System.out`), and explicit *constants* (such as `null` and `"OK"`). For creating object usage models, we use the JADET tool [29].

JADET applies intraprocedural analysis to the program given as an input and uses data flow analysis to discover how each object is being used. It creates an object usage model for each statically identifiable object. An object usage model is essentially a nondeterministic finite automaton with epsilon transitions. Its states are based on the locations in the source code, and its transitions are operations performed by the method that uses the object being modeled. We call these operations *events*. For example, if `x` is a formal parameter of `foo` (and therefore a statically identifiable object associated with `foo`), and `foo` calls `bar(x)`, then calling `bar` is an *event* associated with `x`, because `x` is being passed as an actual argument to `bar`. More precisely, an event associated with an object is one of the following:

- a method call (including constructor calls) with the object being used as the *target* or an *argument*: `x.bar(y, z)` is an event associated with `x`, `y`, and `z`.
- a method call with the object being the value that was *returned:* `x = map.items()` is an event associated with `x`.
- field access with the object being the value that was *read:* `x = System.out` is an event associated with `x`.

Fig. 3 shows an object usage model for the `list` object, as generated by `JADET` from the code shown in Fig. 1. In this model, we use "`foo @ n`" to represent the fact, that the object being modeled was passed as the *n*-th argument to `foo`.

Temporal logic model checking [7] is, in general, a technique for verifying that a given system satisfies the specification given as a temporal logic formula. We use Kripke structures as the model of the system, and CTL (*Computation Tree Logic*) [8] as the language for representing specifications.

Let AP be a set of *atomic propositions*. A Kripke structure over AP is a tuple $M = (S, I, R, L)$, where S is a finite set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a left-total[a] transition relation, and $L : S \to 2^{AP}$ is a labeling function. Atomic propositions are used to describe the state the system is in, and the Kripke structure represents transitions between the states of the system. Because R is a left-total relation, all the behaviors of the system are infinite.

CTL is a temporal logic used to predicate over the behaviors represented by the Kripke structure. It is defined over the same set AP of atomic propositions that the Kripke structure uses:

1) *true* and *false* are CTL formulas
2) Every atomic proposition $p \in AP$ is a CTL formula
3) If f_1 and f_2 are CTL formulas, then so are $\neg f_1$, $f_1 \vee f_2$, $f_1 \wedge f_2$, $f_1 \Rightarrow f_2$, and $f_1 \Leftrightarrow f_2$.
4) If f_1 and f_2 are CTL formulas, then so are $\mathsf{AX}\, f_1$, $\mathsf{EX}\, f_1$, $\mathsf{AF}\, f_1$, $\mathsf{EF}\, f_1$, $\mathsf{AG}\, f_1$, $\mathsf{EG}\, f_1$, $\mathsf{A}[f_1 \mathsf{U} f_2]$, $\mathsf{E}[f_1 \mathsf{U} f_2]$.

A means "for all paths", and E means "there exists a path". X stands for "next", F stands for "finally", G stands for "globally", and U stands for "until". The intuitive meaning of some CTL formulas is as follows: $\mathsf{AX}\, f_1$ means that "for each state $s_0 \in I$, for all (A) paths starting in s_0, f_1 holds in the next (X) state". $\mathsf{EF}\, f_1$ means that "for each state $s_0 \in I$, there exists (E) a path, where f_1 holds somewhere along (F) this path". $\mathsf{AG}\, f_1$ means that "for each state $s_0 \in I$, for all (A) paths starting in s_0, f_1 holds in all states along (G) the path". $\mathsf{A}[f_1 \mathsf{U} f_2]$ means that "for each state $s_0 \in I$, for all (A) paths starting in s_0, f_1 holds until (U) f_2 holds (i.e., f_2 must hold somewhere along the path, and until then, f_1 must always hold). An atomic proposition p holds in a given state iff this state is labeled with p.

Model checking a given CTL formula f against a given Kripke structure $M = (S, I, R, L)$ is equivalent to asking if f holds for each $s_0 \in I$. If it does, f is said to be true for M; if it does not, f is said to be false for M.

[a] $\forall s_1 \in S \,.\, \exists s_2 \in S \,.\, (s_1, s_2) \in R$

Fig. 4. CTL and model checking in a nutshell.

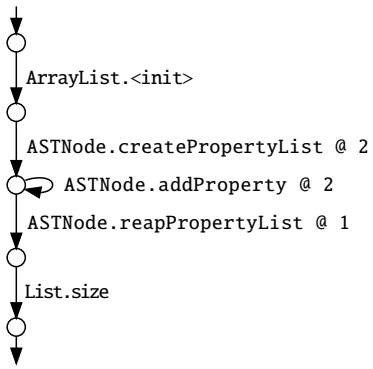

Fig. 3. An object usage model for `list` from Fig. 1.

B. Prefix models

Object usage models provide a representation of the way a single object is being passed to various methods, and we want to discover operational preconditions for each parameter of each method separately. To be able to do this, we must transform each object usage model into a set of *prefix models,* each focusing on a specific combination of a method and its parameter—a specific *target event e* in the object usage model. Since we want to discover *preconditions*, we are only interested in what happens *before e*. Thus, we must prune the model by removing some states and transitions.

In general, pruning gives us a prefix model in which each path starting in the initial state either ends with the transition e to some post-target state, or can be extended to such a path. This makes the prefix model describe the way the object is used under the assumption that the event e always finally happens. This in turn allows us in the end to discover operational preconditions of e.

For most of the models, pruning boils down to removing the states that happen after e. For example, if we take the object usage model shown in Fig. 3 as an input, then the prefix model that focuses on the call to `reapPropertyList` as the target event will simply not have the transition labeled with `List.size`, and the next-to-final state will become the final state. In some cases this is more complicated, but due to space constraints we will not go into detail here.

C. Kripke Structures

In the third step, we transform each prefix model into a *Kripke structure* (see Fig. 4). The problem here is to transform transitions from the prefix model into states in the Kripke structure, and have transitions in the Kripke structure reflect the way transitions in the prefix model can follow one another. We have implemented the solution of Jonsson, Khan, and Parrow [20], who faced an almost identical problem before.

Basically, their idea is that each transition $s_1 \xrightarrow{e} s_2$ is transformed into a separate state in the Kripke structure, and the transition relation is built in such a way that there is a transition between states corresponding to $s_1 \xrightarrow{e_1} s_2$ and $s_3 \xrightarrow{e_2} s_4$ iff $s2 = s3$. Labels are given as follows: State corresponding to the transition $s_1 \xrightarrow{e} s_2$ gets labeled with e. Also, additional initial and "sink" states (both unlabeled) are introduced.[2] Fig. 5 shows a Kripke structure created from a prefix model focused on the call to `reapPropertyList` (this prefix model, which we do not show because of space constraints, was produced from the model shown in Fig. 3) using the procedure described above. The intuitive meaning of the state labeling is "what is the event that has happened most recently?" This allows us later to learn CTL formulas that describe the temporal relation between events the object must go through before participating in the target event.

[2]The "sink" state is needed to make sure that the transition relation is left-total.

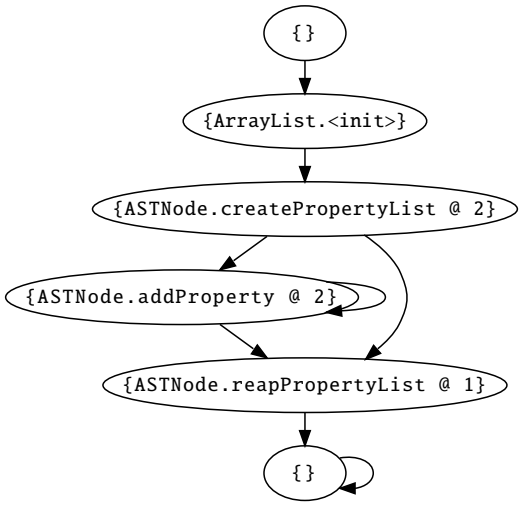

Fig. 5. Kripke structure created from a prefix model focused on the call to `reapPropertyList`. The original model, from which the prefix model stems, is shown in Fig. 3.

III. Extracting CTL specifications

A. Generating CTL formulas

Once we have a Kripke structure, we are able to model check any CTL formula against it (see Fig. 4 for a brief overview of CTL and model checking). Our idea here is to generate *many plausible* CTL *formulas and model check them to discover those that are true*. The most accurate approach would be to generate all possible CTL formulas up to a certain depth.[3] Unfortunately, this is infeasible due to a combinatorial explosion in the number of formulas.

Even assuming that there are only four atomic propositions (as is the case in Fig. 5), and limiting ourselves to only a minimal, complete set of CTL operators (say, the *existential normal form* with only *true*, $\wedge$, $\neg$, EX, EU, and EG operators), the number of CTL formulas increases very rapidly: There are 5 formulas of depth 0, 65 formulas of depth 1, almost 10 000 formulas of depth 2, and over *two-hundred-million* formulas of depth 3. While many of those are equivalent, the number of formulas still remains too large. Therefore, we have decided to limit ourselves to only a set of CTL formulas that can be generated from *predefined templates.* These templates can be easily adapted to the needs of the user. We did not investigate in detail the problem of choosing best templates. Instead, as a proof of concept, we settled on a simple set of templates that would allow Tikanga to learn two things: the fact that a method call must occur, and the fact that a method call may/must occur after another method call. Our proof-of-concept set of templates around atomic propositions (p_i) is as follows:

[3]The depth of a CTL formula is the depth of its abstract syntax tree.

- AF p_1 : The object *must* eventually participate in the event denoted by p_1. For example, AF `createPropertyList @ 2` means that the object *must* eventually be passed as the second argument to `createPropertyList`.
- AG ($p_1 \Rightarrow$ AF p_2) with $p_1 \neq p_2$: Whenever the object participates in the event denoted by p_1, it *must* at some later point participate in the event denoted by p_2. For example, AG (`lock` $\Rightarrow$ AF `unlock`) means that after `lock` has been called, `unlock` *must* be eventually called as well.
- AG ($p_1 \Rightarrow$ EF p_2) with $p_1 \neq p_2$: Whenever the object participates in the event denoted by p_1, it *must be possible* for the object to at some later point participate in the event denoted by p_2. For example, AG (`ArrayList.<init>` $\Rightarrow$ EF `addProperty @ 2`) means that after creating an array list there *exists a path* through the program such that the list is *eventually* passed as the second argument to `addProperty`.

These templates effectively reduce the set of generated CTL formulas. Very long methods, though, can still result in large object usage models and hence tens of thousands of CTL formulas. We therefore limit ourselves to only those atomic propositions, that occur more often than a specified minimum number of times (See the description of *minimum support* in Section IV). This drastically reduces the number of formulas while not influencing our final results.

B. Model checking CTL formulas

Having created a Kripke structure and generated a set of CTL formulas, we now must discover which CTL formulas are true, and which ones are false. We do this using *model checking.* Model checking CTL is particularly easy: We use the algorithm of Clarke, Emerson, and Sistla [7]. It treats the CTL formula as a tree (with subformulas as subtrees), and then travels up the tree and model checks each subformula in each state of the Kripke structure. In the end, if the whole formula holds for all initial states of the Kripke structure, it is marked as being true.

This algorithm not only has polynomial complexity, it also facilitates reuse of the results obtained earlier. Some of the subformulas occur in more than one formula, and in that case they do not need to be reevaluated. These are very important properties, that allow us to achieve a very good run-time performance, and thus scalability to real-life programs.

One important thing to be mentioned here is that we do not attempt to remove related CTL formulas, such as (3) and (4) in Fig. 7. It is clear that formula (3) being true implies formula (4) being true as well, and this can be seen as an argument for keeping only formula (3). However, in our case this would result in crucial information being deleted, because we are ultimately interested in CTL formulas that hold across many Kripke structures. Keeping only the strongest formula in each case could lead to missing weaker CTL formulas that are, however, true for many more Kripke structures than the strongest ones. This would in turn distort operational preconditions found, and this is something we want to avoid.

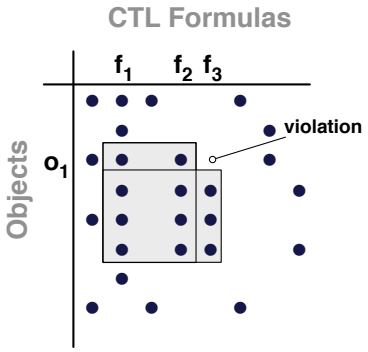

Fig. 6. Detecting violations via concept analysis. Each rectangle corresponds to an operational precondition; gaps indicate potential violations [22]. TIKANGA creates multiple concept analysis matrices: one for each formal parameter of each method. Objects are actual parameters used by the callers of the method.

IV. LEARNING OPERATIONAL PRECONDITIONS

After CTL formulas for all Kripke structures have been generated and model checked, TIKANGA can start learning operational preconditions for all parameters of all methods called in that program. For a given parameter p of a method M, TIKANGA looks for sets of CTL formulas that are *true for many objects* passed as p to M (this boils down to finding CTL formulas that are true for the corresponding Kripke structures). Each such set of CTL formulas forms an operational precondition (OP). Thus, each OP not only represents *what* needs to be done, but also a temporal relation between events that has to be preserved.

TIKANGA needs to decide whether the CTL formulas that hold for the actual argument of M are really relevant or just *noise*—that is, infrequent temporal relations that are particular to specific callers. Our solution to this problem is as follows: If a particular set of CTL formulas occurs frequently (i.e., holds for several objects passed to M as the same actual argument), then all elements in this set are relevant and, in fact, constitute an OP of this parameter of M. For example, if there are many calls to `reapPropertyList`, but only for a few objects being passed to it the formula `AF flagProperties @ 1` holds, then this particular CTL formula will be discarded as not being part of true OPs. This learning by eliminating noise is done by *formal concept analysis.*

Concept analysis is, broadly speaking, a technique for finding *patterns* [16]. It takes as its input a set of conceptual objects, a set of conceptual properties, and a cross table associating objects with properties. Fig. 6 shows a sample cross table with rows being conceptual objects and columns being conceptual properties. Concept analysis produces all concepts found in the cross table, where a concept is a set of objects A and a set of properties B such that every object in A is associated with all properties from B and sets A and B are *maximal,* i.e., it is not possible to add elements to one set without influencing the second one. Intuitively, a concept is a rectangle (not necessarily contiguous) in the cross table: Fig. 6 shows two such rectangles.

We use the COLIBRI tool [22] for formal concept analysis, with conceptual objects being statically identifiable objects; and conceptual properties being CTL formulas. For each parameter of each method called in the program we create a separate cross table such that we can focus only on objects passed to that particular method as that particular parameter. We also introduce a configuration parameter called *minimum support*, which puts a lower boundary on the number of objects in a concept. This makes sure that COLIBRI reports sets of CTL formulas that are common to at least *minimum support* many statically identifiable objects passed to the method; it thus eliminates noise caused by infrequent usage.[4] The resulting sets are operational preconditions for this specific parameter of this specific method.

As an example of an operational precondition, consider the one shown in Fig. 7 for the target object of a call to `Label.place`. This operational precondition has been extracted by TIKANGA from ASPECTJ. The code that this operational precondition stems from is responsible for generating bytecode instructions, amongst them `goto` statements, that jump unconditionally to a specific label. The operational precondition that TIKANGA has extracted contains the following rules:

1) A label is always created by calling its constructor before being placed (which is not that obvious, as there may also be a factory method to be used instead) (1)
2) After a label has been created, it *can* be used as the target of a goto statement (2)
3) After a label is created, it is always placed (3, 4)
4) After a label is used as the target of a goto statement, it is always placed (5, 6)

Note that ASPECTJ apparently may produce labels that are not referenced by a goto statement; we do not see this as a problem.

V. DETECTING USAGE ANOMALIES

A. Discovering violations

As its last step, TIKANGA looks through all the objects used as actual arguments and identifies those that violate the operational preconditions found. Consider the operational precondition shown in Fig. 7. It is clear that if some `label` is first being created, then used as a reference point for a goto and finally placed, such a code satisfies this operational precondition. But if, say, a call to `CodeStream.goto_` was entirely missing, we would say that such a usage violates the operational precondition. Generally, we can say that *if there are* CTL *formulas that are present in the OP, but do not hold for the object passed to the method, the object violates the operational precondition*.

However, we need to weaken the condition given above a bit. After all, if the operational precondition is violated by many objects, it is rather because the operational precondition

[4]We also use the minimum support to avoid generating concept analysis matrices for events that happen too rarely, and to avoid generating CTL formulas that contain rarely occurring atomic propositions (see Section III-A).

AF Label.<init> (CodeStream)	(1)
AG (Label.<init> (CodeStream) ⇒ EF CodeStream.goto_ (Label) @ 1)	(2)
AG (Label.<init> (CodeStream) ⇒ AF Label.place ())	(3)
AG (Label.<init> (CodeStream) ⇒ EF Label.place ())	(4)
AG (CodeStream.goto_ (Label) @ 1 ⇒ AF Label.place ())	(5)
AG (CodeStream.goto_ (Label) @ 1 ⇒ EF Label.place ())	(6)

Fig. 7. Operational precondition for the target object of a call to `Label.place`. See Section IV for a discussion.

is too restrictive, than because all those objects are in an incorrect state when being passed to the method. We deal with this by introducing a parameter called *minimum confidence*, which is a number between zero and one, and is defined as follows: Let s_{OP} be the support of the operational precondition (i.e., the number of objects that adhere to it), and s_{PV} be the support of the potential violation (i.e., the number of objects that violated the operational precondition in the same way). Confidence is then given by the formula $c = s_{OP}/(s_{OP}+s_{PV})$. Just as we used minimum support to ensure that only frequently occurring sets of CTL formulas get reported as operational preconditions, we use minimum confidence to ensure that only infrequently occurring divergences from operational preconditions get reported as real violations.

To detect violations in the way described above, we again use Colibri [22]. Detecting violations is equivalent to detecting "gaps" in the concept analysis cross table. In Fig. 6, we can see that the object o_1 violates the operational precondition represented by the set of CTL formulas $\{f_1, f_2, f_3\}$, because the formula f_3 does not hold for it. We can also say that the confidence of this violation is 0.75, because the support of the operational precondition is 3 ($\{f_1, f_2, f_3\}$ all hold for three objects, so $s_{OP} = 3$) and there is only one violation of the operational precondition where only f_3 is missing (so $s_{PV} = 1$). Thus, $c = s_{OP}/(s_{OP} + s_{PV}) = 3/(3 + 1) = 0.75$. Detecting violations in this way was introduced by Lindig [22] and used in our previous work on detecting object usage anomalies [29]. In the context of this paper, the important thing is that we can find operational preconditions and their violations in a way that is both effective and efficient.

B. Filtering violations

The approach described above leads to discovering a large number of violations of operational preconditions, most of them being false positives. We deal with this problem by first filtering violations and then ranking the remaining violations, so that the user can focus only on the top-rated ones.

When it comes to filtering, we first filter away all violations of operational preconditions for `java.lang.String`, `java.lang.StringBuffer`, and `java.lang.StringBuilder` classes. From our experience, these are almost always false positives. We believe that this is because most methods of these classes are used very often in many different ways, and yet do not really have any preconditions. This confuses the analysis into coming up with many varied operational preconditions (all, or almost all of them bogus) that are then very often violated. The same problem could in principle occur for user-defined classes, and theoretically it should be possible to automatically filter out all violations for certain classes, if there are too many such violations. However, we have not encountered any user-defined classes with such characteristics[5], and so we decided just to filter away the three classes listed above.

As the second filtering step, we leave only one violation for each statically identifiable object. While this may lead us to miss some true positives, more often than not it results in simply removing duplicates.

C. Ranking violations

For ranking violations, we compare their *lift* measures. Formally, we can treat each violation of an operational precondition as a violation of an *association rule* $A \rightarrow B$, where A and B are sets of CTL formulas, $A \cup B$ is the set of formulas that constitutes the operational precondition, and B is the set of CTL formulas that do not hold for the object violating the operational precondition. Lift of an association rule (and thus, of a violation in our setting) is defined as follows: Let s_{AB} be the support of the operational precondition (i.e., the number of times CTL formulas from the set $A \cup B$ occur together), s_A be the support of the set A of CTL formulas, s_B be the support of the set B of CTL formulas, and n be the number of times the target event of the operational precondition occurs in the program analyzed. Lift is then given by the formula $lift(A \rightarrow B) = (s_{AB}/(s_A * s_B)) * n$. This formula measures how many times more often A and B occur together than expected if they were statistically independent. We rank all violations according to their lift measures, and in the end remove those, whose lift value is equal to one (i.e., those, where A and B seem independent).

VI. Evaluation

To evaluate the effectiveness of Tikanga, we have applied it to several complex Java projects (see Table I): Act-Rbot, a cognitive agent based social simulation toolkit, Apache Tomcat, a servlet container, ArgoUML, a UML design tool with cognitive support, AspectJ, an aspect-oriented extension to the Java programming language, Columba, an email client, and Vuze, a bittorrent client.

[5]This does not mean there are no such classes in the projects we have analyzed. It may well be that they are just used too rarely for Tikanga to try to learn about their intended usage

TABLE I
DETAILS OF TIKANGA CASE STUDY SUBJECTS.

Program	Origin	# Classes	# Methods	# Violations	Total time
ACT-RBOT 0.8.2	http://www.acis.nl/researchdocs/	344	3 401	23	2:53
APACHE TOMCAT 6.0.18	http://tomcat.apache.org/	1 462	16 347	18	4:31
ARGOUML 0.26	http://argouml.tigris.org/	1 897	13 824	194	6:18
ASPECTJ 1.5.3	http://www.eclipse.org/aspectj/	2 957	36 045	189	11:36
COLUMBA 1.4	http://www.columbamail.org/drupal/	1 488	8 590	19	1:45
VUZE 3.1.1.0	http://azureus.sourceforge.net/	5 532	35 363	304	13:29

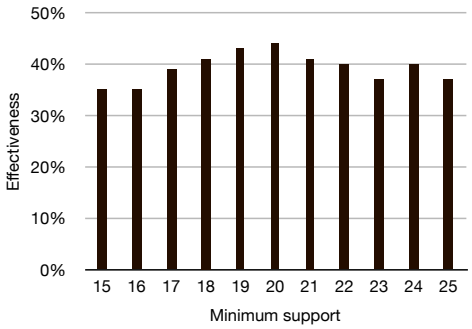

Fig. 8. Influence of minimum support on TIKANGA's effectiveness on ASPECTJ (minimum confidence fixed at 0.9, top 25% violations investigated).

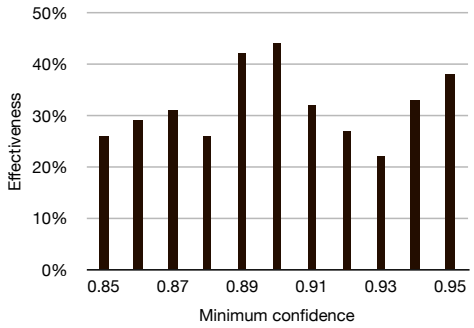

Fig. 9. Influence of minimum confidence on TIKANGA's effectiveness on ASPECTJ (minimum support fixed at 20, top 25% violations investigated).

We have used 20 as the minimum support threshold and 0.9 as the minimum confidence threshold. We have also focused on investigating only top 25% violations.[6] These constants were obtained by empirically testing a few alternatives for ASPECTJ in order to balance the number of false positives and true negatives, but we have not investigated if this combination is optimal. However, we have done some experiments to measure how small changes to all of those parameters influence the results for ASPECTJ. Results of those experiments can be seen in Fig. 8, Fig. 9, and Fig. 10. We can see that effectiveness is fairly insensitive to small changes to minimum support, and that it is sensitive to small changes to minimum confidence. Effectiveness is also sensitive to the number of top violations investigated. One interesting artifact is that effectiveness rises to a certain point, indicating that most interesting violations are not at the top, but near it. Another artifact is that effectiveness seems to stabilize at the level of around 25%, which is incidentally the same value as overall effectiveness for ASPECTJ (cf. section VI-A). This suggests that, after a certain point, distribution of true positives is quite uniform.

Table I contains a list of our case study subjects, including their size (number of classes, number of methods), and a short summary of our results (number of violations reported by the tool and the time needed to perform the analysis[7]).

[6]The only exception is ASPECTJ, for which we have investigated all violations; see section VI-A.

[7]The time reported encompasses all steps that TIKANGA needs to extract violations from a program, including the time spent by JADET on creating object usage models.

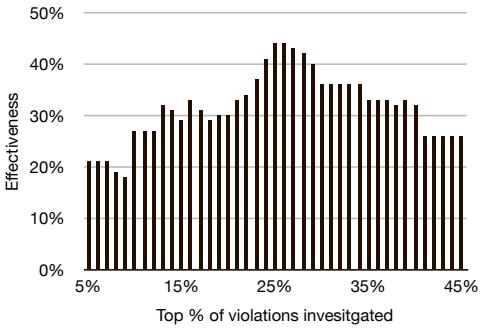

Fig. 10. Influence of the number of violations investigated on TIKANGA's effectiveness on ASPECTJ (minimum support fixed at 20, minimum confidence fixed at 0.9).

A. Case study: AspectJ

The largest program in our set is ASPECTJ, a compiler for the ASPECTJ language. ASPECTJ is a sufficiently complex, big, and mature project to put our technique to a good test. TIKANGA reported 189 violations of operational preconditions in ASPECTJ. Each reported violation contained the following information: the operational precondition being violated, the object that violates the precondition, and the set of CTL formulas that do not hold for the object (but are part of the operational precondition). Note that TIKANGA not only reports violations, *but also shows what is missing:* If the violation is indeed a defect, it shows how to fix it. We inspected those 189 violations manually and classified them into three categories:

```
for (Iterator it = c1.iterator();it.hasNext();){
    E e1 = (E) it.next();
    ...
    for (Iterator it2 = c2.iterator();it.hasNext();){
        E e2 = (E) it2.next();
        ...
    }
    ...
}
```

Fig. 11. In AspectJ, an inner loop checks the iterator of the outer loop.

Defects. This category is self-explanatory, but there is one important point we want to make here. It sometimes happens that there is a method that violates the contract of its base class, but the application itself does not fail because of this. One example is if a comment in the base class states that a particular method accepts `null` values passed as an argument, but the implementation of that method in the derived class does not. If this method is public, we still mark it as defective, because someone may cause it to fail by following the contract of the base class.

Code smells. This category contains all violations that are not defects, but the violating methods have properties indicating that something may go wrong [14] or they might be improved in a way that improves readability, maintainability or performance of the program. An example might be a method that uses a `for` loop to iterate through a collection and breaks unconditionally out of the first iteration. If the collection can have at most one element, this code will work, but it cannot be treated as fully correct.

False positives. This category contains all violations that are neither defects nor code smells.

Out of the 189 violations reported for AspectJ, we categorized 11 as defects, 36 as code smells, and the rest as false positives. Hence, 25% of all the violations are worth investigating. Also, all violations are unique, i.e., each object violating an operational precondition appears only once.[8]

Our ranking method proved to be very effective in ordering true positives higher on the list. Out of top 50 violations (≈ 25%) reported for AspectJ, 3 are defects, and 19 are code smells. This means that 44% of the top-ranked violations are actual code issues, resulting in a 44% true positive rate for Tikanga.

Let us present some of the most interesting violations found. Out of 9 unique defects found by Tikanga in AspectJ, two are severe enough to cause a compiler crash (previously reported as bugs #218 167 and #218 171 in the AspectJ bug database). Both are simple typos occurring in two different methods; they violate operational preconditions of two different objects each. In each case, both affected objects are instances of the `Iterator` class, and in both cases operational precondition of the `Iterator.next` method were violated. The skeleton of the defective code is shown in Fig. 11. Iterator `it` violates the operational precondition, because after calling `hasNext` there is no guarantee that `next` will be called.[9] Iterator `it2` violates the operational precondition, because `hasNext` is not being called before `next` is called.

Four defects are located in methods that violate the contract of the method they override. All of them take a progress monitor instance as one of the parameters; in all cases, the overridden method says this instance may be `null`. One of the operational preconditions of the method `IProgressMonitor.done` states that the monitor should be the return value from the factory method `Policy. monitorFor`. It turns out that this very method handles `null` by returning an instance of the class `NullProgressMonitor`. This solves the problem of the monitor being `null`. Since the defective methods do not have this call and do not check explicitly for `null`, they throw a `NullPointerException` if they are called with `null` as the progress monitor.

Analyzing AspectJ from creating object usage models to discovering violations took less than 12 minutes on a 1.83 GHz Intel Core Duo machine. We have investigated all 189 violations manually, looking at each of them and deciding whether the violation is a defect, a code smell, or a false positive. The categorization process took us altogether about twenty hours, which is about 6 minutes per violation on average. We were able to dismiss many violations as false positives quickly thanks to Tikanga giving us rough information on what the fix should achieve (the false CTL formulas). On the other hand, some violations were difficult enough to require half an hour or more of investigation. When we were unsure if the code was correct and were unable to trigger incorrect behavior, we classified it as correct and the violation as a false positive to make sure that our reported precision rate is not an overestimation of what an expert in the project would achieve.

B. Other case study subjects

In addition to AspectJ, we have applied Tikanga to several other projects (see Table I). For all those projects we only investigated the top 25% of reported violations. A summary of the results of those investigations can be found in Table II. For each of the projects, we present the total number of violations, the number of violations top 25% amounts to (these are the violations we investigated[10]), the number of defects, code smells, and false positives found by investigating them, and the effectiveness (i.e., the percentage of violations that were defects of code smells).

Fig. 12 shows the skeleton of one of the code smells found by Tikanga in Columba. The problem with this code is that it sets the layout for the main panel twice: the first time in the

[8]Despite this, we found two defects that got reported twice. The reason was that the defect distorted the usage pattern of two different objects at the same time. This means we found 9 *unique* defects in AspectJ.

[9]In theory, the second loop could be an infinite one; this is the exact semantics of CTL.

[10]Sometimes this is more than exactly 25%. The reason for this is that some violations have the same ranking, so we had to include all such equally-ranked violations.

TABLE II
SUMMARY OF THE RESULTS FOR THE EXPERIMENT SUBJECTS. (SEE SECTION VI-B FOR A DISCUSSION.)

Program	# Violations		# Defects	# Code smells	# False positives	Effectiveness
	Total	Top ≈25%				
ACT-RBOT 0.8.2	23	9	0	6	3	**67%**
APACHE TOMCAT 6.0.18	18	4	0	1	3	**25%**
ARGOUML 0.26	194	67	3	8	56	**16%**
ASPECTJ 1.5.3	189	50	3	19	28	**44%**
COLUMBA 1.4	19	5	0	4	1	**80%**
VUZE 3.1.1.0	304	77	2	15	60	**22%**

```
JPanel mainPanel = new JPanel(new BorderLayout());
...
FormLayout layout = new FormLayout(...);
...
mainPanel.setLayout(layout);
```

Fig. 12. In COLUMBA, a panel gets its layout set twice, and differently at that.

```
public Object clone() {
    FigObject figClone = (FigObject) super.clone();
    Iterator it = figClone.getFigs().iterator();
    figClone.setBigPort((FigRect) it.next());
    figClone.cover = (FigRect) it.next();
    figClone.setNameFig((FigText) it.next());
    return figClone;
}
```

Fig. 13. Code that we classified as a false positive in ARGOUML. See Section VI-B for a discussion.

constructor call, and the second time in the call to `setLayout`. This code will not fail, but it is confusing, especially since the layouts used are different. In the end, the panel will use the layout specified as the argument to `setLayout`, but it is easy to miss this call and think that border layout will be used.

Again, we stayed on the conservative side when categorizing violations found by TIKANGA. Consider the code from ARGOUML shown in Fig. 13. This is the code of the `clone` method inside the `FigObject` class. One problem with this code is that it assumes that there are three "figs" and that they are stored in the given order. A much better approach is demonstrated in the base class of `FigObject`: iterate through the "figs", take a look at what they are, and then set an appropriate field or call an appropriate method. However, we still marked this code as a false positive, because during normal operation it will work fine, is consistent with what is in the constructor, and is easy to understand. This has had a huge influence on the overall accuracy of TIKANGA on ARGOUML, because there are 27 similar violations for various classes in the same class hierarchy reported by TIKANGA. Classifying all of them as code smells would increase our true positive for top violations in ARGOUML from 16% to 57%.

Overall, the top violations reported by TIKANGA for the projects we analyzed contained 8 defects, 53 code smells, and 151 false positives. This gives an overall effectiveness of 29%. However, this result is strongly influenced by projects that had lots of violations reported, with most of them being false positives (cf. our discussion of ARGOUML above). The *average effectiveness per project* is much higher 42%—i.e., on average, 42% of the top violations reported for a project by TIKANGA are true positives.

C. Limitations and threats to validity

The most important limitation of our approach is that it needs a substantial code base to learn from. While this can be partially circumvented (e.g., if one wants to use some library and wants TIKANGA to check if one is not making any mistakes, one can use someone else's program to learn from), it is an unavoidable price for the ability to tap into developers' knowledge and experience that is contained in those code bases. Also, TIKANGA is only useful for single-threaded programs, but it can handle the whole JAVA language, including exception handling.

Another limitation is that we are restricted to only those CTL formulas that fit into the prescribed templates. This is partially alleviated by the fact that the templates can be easily changed and adapted to the user's needs, but it still requires the user to do some work before getting the most out of the tool. In fact, we used only very simple templates to illustrate the technique. It is perfectly possible (and even expected) that more interesting operational preconditions and violations can be found if one chooses the templates more carefully.

It would be very useful to learn operational preconditions from one project or set of projects and look for violations in another one (such separation into training and testing sets is common in machine learning, for example). Such a possibility would allow the user to use some sort of training code, for instance to learn proper API usage of commonly used library functions. TIKANGA does not support this directly, but in principle it is possible to use TIKANGA to learn operational preconditions from the training set, and learn true CTL formulas from the testing set. It is then quite straightforward to write a script that investigates the CTL formulas true for the testing set and checks them against operational preconditions.

We have identified the following threats to validity:

- We have investigated six programs with different application domains, sizes, and maturity, and our tool was always able to discover issues with the program at hand. However, these results may not generalize to arbitrary projects; proprietary or closed source programs may have very different properties.

- The tools we have used (JADET and Colibri) could be defective. We think this is very improbable, especially for Colibri, whose implementation is publicly available [22]. We have thoroughly used and validated the JADET and Tikanga code, so we believe that any defects left affect only a small number of violations, and thus do not spoil the results overall.
- The results of the categorization process performed on violations depend on the expertise of the human applying the approach. However, if anything, this would *improve* our results—because we have marked violations as defects only if we were completely sure that they are indeed defects (e.g., by crashing the program, making sure the contract was violated, seeing the code changed in some later version in the way suggested by Tikanga, etc.). An experienced developer may spot potential problems where we see false positives.

VII. Related Work

To the best of our knowledge, the present work is *the first to take an operational view at preconditions*—learning and checking what needs to be done to call a function, and *the first to learn* CTL *specifications directly from program code, instead of requiring them to be provided by the user*. However, learning CTL formulas from Kripke structures has been done before by Chan [6]. Chan introduced the concept of so-called temporal-logic queries, which are similar to our templates, but restricted to only one placeholder. This work was extended later by Gurfinkel, Chechik, and Devereux to queries with multiple placeholders [17]. Temporal-logic queries with multiple placeholders are very close to our templates, but the technique of solving them is not directly applicable in our setting: Solutions to queries are the strongest possible formulas, whereas we need all possible formulas in order to be able to look for patterns later on (see Section III-B).

When it comes to learning from existing code in general, or to automatic defect detection, there are many relevant existing approaches. Due to space constraints, we will just take a closer look at some of them.

A. Learning from code

Ernst et al. [12] have written the seminal work on inferring invariants dynamically using DAIKON. DAIKON examines program run traces and learns formulas such as $1 < x < 5$ about variables. Thus, it is among others able to learn what we call in this paper "axiomatic preconditions". Similarly-themed work was done by Hangal and Lam [18]. Their DIDUCE learns invariants at run-time while simultaneously detecting and reporting violations.

Cook and Wolf [9] wrote the seminal work on learning models of software development processes. They create finite state automata based on traces of events, using techniques such as grammar inference, neural networks, and Markov models. Ammons, Bodík, and Larus [3] use dynamic analysis of C programs to create so-called specifications automata. They use a probabilistic finite state automaton learner for that purpose. Alur, et al. [2] analyze definitions of classes (as opposed to their clients) to come up with interface specifications for them. Their approach is based on gradual refinement of the specifications using model checking. Shoham et al. [28] presented an approach for mining finite machine specifications for classes using sophisticated static analysis techniques. Javert by Gabel and Su [15] is a tool that uses an interesting approach of learning so-called micropatterns from runtime traces and then combining them into larger specifications depicted as finite state automata. Acharya et al. [1] apply model checking to learn specifications. In principle, going from finite state automata produced by all those techniques to CTL formulas should be possible using the approach described in this paper. This would allow applying those techniques for defect detection purposes.

The approach of Ramanathan et al. [25] learns axiomatic preconditions and constraints on method calls of the form "a call to `g` is always preceded by a call to `f`". The specifications produced are fairly expressive, but the simple ordering used is much weaker than what CTL formulas can express. They used their approach to find defects, too, but did not report on the rate of false positives. The Perracotta tool by Yang and Evans [32] mines temporal rules of program behavior fitting into templates (such as alternation) provided by the user. The Perracotta templates are limited to occurrences of method calls, and thus do not take into account the flow of objects between arguments of different methods.

Some research tools support programmers by providing them with examples of API usages, focusing a different problem than Tikanga. MAPO by Xie and Pei [31] provides sample sequences of method calls, but does not relate them to objects. MAPO provides information about the order in which the methods are typically called, but not how objects flow through them. Prospector by Mandelin et al. [24] and XSnippet by Sahavechaphan and Claypool [27] can provide information on how to create an object of a given type. These tools do not address the finer problem of how a method's arguments should be constructed.

B. Automatic defect detection

Out of the abundance of work on automatic defect detection, let us present some highlights. Engler et al. [11] use static analysis to learn common behavior, and find and report to the user uncommon behavior, with the assumption that it points to a defect. This is the same assumption we use when creating CTL specifications and looking for their violations; however, Engler et al.'s approach needs a checker written by the user to direct its analysis. We need CTL formulas' templates provided by the user, but these are much more generic (and thus applicable to a wider range of potential problems) than a checker can be. FindBugs by Hovemeyer and Pugh [19] is equipped with a fixed lists of "defect patterns" and checks the code looking for places where those are violated.

A lot of work focuses on detecting code that violates a *given* specification. Testing [4], static analysis [5], [13], and model checking [26] have been used for that purpose.

These approaches are typically very effective and precise when looking for violations, but the specification has to be provided by the user and this is the main weakness. Creating specifications for project-specific classes is a lot of work that cannot be reused in other projects, and this is the problem we are addressing with TIKANGA.

The research area that is most related to our work on detecting violations of CTL formulas is on using code [21], [29], traces [10], [30], and software repositories [23] to learn rules and then check the program for conformance with these rules. Two approaches that are closest to TIKANGA is Li and Zhou's PR-MINER [21], and our previous work on JADET [29].

PR-MINER uses frequent itemset mining to find sets of functions, variables, and data types that frequently appear together. TIKANGA uses a very similar concept, but on CTL formulas. CTL formulas are much stronger that simple occurrence patterns, and this allows us to detect a much broader range of defects (e.g., the two defects that crash ASPECTJ could not be detected by PR-MINER, because calls to `hasNext` and `next` are present; it is just that they are not used the way they should be).

JADET, our own earlier work, is a tool for detecting object usage anomalies. TIKANGA uses object usage models produced by JADET and applies concept analysis to learn patterns and find violations, but that is where the similarities end. JADET focuses on *whole usage patterns* instead of preconditions. This allows it to learn patterns that are associated with many objects, but at the cost of being more general (and thus less specific) than TIKANGA. What is even more important, JADET's temporal properties are simple orderings of method calls, while TIKANGA learns CTL formulas, which are much more expressive, and thus allow us to detect a much broader range of defects. When it comes to effectiveness, TIKANGA is much more effective than JADET (with 25% vs. 11% true positive rate for all violations reported for ASPECTJ).

VIII. CONCLUSIONS AND CONSEQUENCES

In modern object-oriented programs, most complexity stems not from within the methods, but from method compositions. TIKANGA both addresses and leverages this complexity. It learns from actual method usage and detects deviant temporal relations between method calls, thus automatically discovering a surprisingly high number of both subtle and blatant defects in well-tested production code. It is also, to the best of our knowledge, *the first approach to learning* CTL *formulas*. The approach is lightweight and easily scales to large bodies of code. However, there is still much room for improvement. Our future work will focus on the following topics:

Procedural languages. Parameters are part of every programming language, and therefore our approach easily extends to other languages. C and C++ programmers, for instance, could benefit from operational preconditions that apply to integer values: These are ubiquitous, and frequently are handles of specific objects, such as files or resources, and not really numbers. The fact that their type is `int` makes them not as easily distinguishable as objects in JAVA. Operational preconditions for integer parameters could be of great help in understanding how to use a particular function.

Different analysis techniques. Right now, we use JADET as the initial static analysis phase, and JADET's analysis is intraprocedural. One may argue that going interprocedural would give further advantages, such as considering what happens inside a function call. Unfortunately, breaking this abstraction barrier brings risks in our context. Consider the code

```
Transition t = TransitionFactory.makeTransition();
addTransition(t);
```

where `makeTransition`, among others, eventually invokes the `Transition` constructor—which we can find out via interprocedural analysis. We could thus make the `Transition` constructor an operational precondition for `addTransition`. This would be an error, though, for `TransitionFactory` is there so that the user should use exclusively it for creating `Transition` objects.

Alternative abstractions. Currently, we use an abstraction based almost wholly on method calls, since this is what the first stage of the analysis gives us. There might be other *abstractions* of objects' properties, that could then later be used as atomic propositions in CTL formulas. One can think of "deeper" callees, as inferred from interprocedural analysis, or common type transformations. Such considerations can be investigated in more detail by replacing JADET with other techniques that do some kind of program analysis.

Early programmer support. Once mined, operational preconditions can be easily integrated into the programming environment, warning the user about potential problems before they become serious. Likewise, operational preconditions can become part of the documentation, specifying important temporal properties that have to be satisfied. The mined CTL formulas can also be checked against given ones, giving additional opportunities for validation.

Useful CTL *templates.* In this work we have chosen a rather simple set of CTL templates, and yet obtained promising results. It would be interesting to do some research in this area and try to identify templates that are most effective in defect detection and still keep the true positive rate as high as possible.

For future and related work regarding TIKANGA, see

http://www.st.cs.uni-saarland.de/models/

Acknowledgments. We are thankful to Sebastian Hack, Łukasz Kaiser, and Frank Padberg for inspiring discussions. Peggy Cellier suggested using lift for ranking anomalies. Valentin Dallmeier, Kim Herzig, David Schuler, and anonymous reviewers provided helpful and constructive comments on earlier revisions of this paper.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *ESEC-FSE 2007: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY: ACM, 2007, pp. 25–34.

[2] R. Alur, P. Černý, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for Java classes," in *POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY: ACM, 2005, pp. 98–109.

[3] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *POPL 2002: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY: ACM, 2002, pp. 4–16.

[4] S. Antoy and D. Hamlet, "Automatically checking an implementation against its formal specification," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 55–69, Jan. 2000.

[5] E. Bodden, P. Lam, and L. Hendren, "Finding programming errors earlier by evaluating runtime monitors ahead-of-time," in *SIGSOFT 2008/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. New York, NY: ACM, 2008, pp. 36–47.

[6] W. Chan, "Temporal-logic queries," in *Proceedings of the 12th International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science 1855. Berlin: Springer-Verlag, 2000, pp. 450–463.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[8] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proceedings of the Workshop on Logics of Programs*, ser. Lecture Notes in Computer Science 131. Berlin: Springer-Verlag, 1982, pp. 52–71.

[9] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 215–249, Jul. 1998.

[10] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," in *ECOOP 2005: Proceedings of the 19th European conference on object-oriented programming*, ser. Lecture Notes in Computer Science 3586. Berlin: Springer-Verlag, 2005, pp. 528–550.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," in *SOSP 2001: Proceedings of the 18th ACM Symposium on Operating Systems Principles*. New York, NY: ACM, 2001, pp. 57–72.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[13] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 2, pp. 1–34, Apr. 2006.

[14] M. Fowler, *Refactoring. Improving the design of existing code*. N.p.: Addison-Wesley, 1999.

[15] M. Gabel and Z. Su, "Javert: fully automatic mining of general temporal properties from dynamic traces," in *SIGSOFT 2008/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. New York, NY: ACM, 2008, pp. 339–349.

[16] B. Ganter and R. Wille, *Formal concept analysis: mathematical foundations*. Berlin: Springer-Verlag, 1999.

[17] A. Gurfinkel, M. Chechik, and B. Devereux, "Temporal logic query checking: A tool for model exploration," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 898–914, Oct. 2003.

[18] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *ICSE 2002: Proceedings of the 24th International Conference on Software Engineering*. New York, NY: ACM, 2002, pp. 291–301.

[19] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *OOPSLA 2004: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY: ACM, 2004, pp. 132–136.

[20] B. Jonsson, A. H. Khan, and J. Parrow, "Implementing a model checking algorithm by adapting existing automated tools," in *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, ser. Lecture Notes in Computer Science 407. Berlin: Springer-Verlag, 1990, pp. 179–188.

[21] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY: ACM, 2005, pp. 306–315.

[22] C. Lindig, "Mining patterns and violations using concept analysis," Saarland University, Software Engineering Chair, Technical Report, 2007, avaliable from http://www.st.cs.uni-saarland.de/publications/; the software is available from http://code.google.com/p/colibri-ml/.

[23] B. Livshits and T. Zimmermann, "DynaMine: Finding common error patterns by mining software revision histories," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY: ACM, 2005, pp. 296–305.

[24] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *PLDI 2005: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. New York, NY: ACM, 2005, pp. 48–61.

[25] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in *PLDI 2007: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY: ACM, 2007, pp. 123–134.

[26] S. P. Reiss, "Specifying and checking component usage," in *AADEBUG 2005: Proceedings of the sixth international symposium on Automated analysis-driven debugging*. New York, NY: ACM, 2005, pp. 13–22.

[27] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for sample code," in *OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY: ACM, 2006, pp. 413–430.

[28] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 651–666, Sep. 2008.

[29] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *ESEC-FSE 2007: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY: ACM, 2007, pp. 35–44.

[30] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *TACAS 2005: Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science 3440. Berlin: Springer-Verlag, 2005, pp. 461–476.

[31] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *MSR 2006: Proceedings of the 2006 International Workshop on Mining Software Repositories*. New York, NY: ACM, 2006, pp. 54–57.

[32] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *ICSE 2006: Proceedings of the 28th international conference on Software engineering*. New York, NY: ACM, 2006, pp. 282–291.