SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2012; **00**:1–16
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Practical JFSL verification using TACO [†]

M. Chicote[1*], D. Ciolek[1] and J. P. Galeotti[2‡]

[1] *Departamento de Computación, FCEyN, UBA, Buenos Aires, Argentina*
[2] *Computer Science - Saarland University, Germany*

SUMMARY

TACO is a SAT-based tool for bounded verification of Java programs. One challenge many formal tools share is to provide a practical interface for a non proficient user. In this article we present an Eclipse plug-in for the static verifier TACO. This plug-in allows a user to walk a counterexample trace mimicking a debugging session. TacoPlug (our plug-in) uses and extends TACO to provide a better debugging experience. TacoPlug interface allows the user to verify an annotated software using the TACO verifier. If TACO finds a violation to the specification, TacoPlug presents it in terms of the annotated source code. TacoPlug features several views of the error trace to facilitate fault understanding. It resembles any software debugger, but the debugging occurs statically without executing the program. Furthermore, should a dynamic analysis be required, TacoPlug presents the user with a unit test case generated by TACO based on the detected violation. We show the usability of our tool by means of a motivational example taken from a real-life software error. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Static analysis; bounded verification; Eclipse plug-in; TACO; test case generation

1. INTRODUCTION

Programmers no longer deal just with a few hundred lines of code but with several hundred million lines controlling some of the most complicated systems ever created. As software presence grows from desktop computers to embedded components such as home appliances, personal digital assistants, medical equipment, etc., an undesirable companion arises: software defects, most commonly known as *software bugs*. Software failures range from those that we may consider annoying to those with more tragic consequences.

The increasing importance of software quality in economy and in everyday life demands the development of (a) more robust engineering techniques and processes to build software, and (b) more advanced tools to help programmers achieve a greater quality in the software artefacts they produce. Modern compilers benefit from program analysis techniques such as type checking and data-flow analysis to warn programmers about unintentional mistakes. In both cases, not only the degree of automation is extremely high, but also advanced IDE's allow programmers to easily understand the problem.

Other approaches allow a procedure in a conventional object oriented language to be automatically checked against a rich interface specification. In this sense, Bounded Verication [7] is a technique in which all executions of a procedure are exhaustively examined within (a) a finite

[*] Correspondence to: Intendente Giraldes 2160 - Ciudad Universitaria - C1428EGA - Buenos Aires - Argentina. E-mail: mchicote@dc.uba.ar
[†] Extended version of the article ”TacoPlug: An Eclipse Plug-In for TACO” presented on TOPI 2012
[‡] On leave from CONICET, Argentina

space given by a bound on the size of the heap and (b) the number of loop unrollings. The scope of analysis is examined in search of an execution trace which violates a given specication.

Several bounded verications tools [3, 4, 8, 9, 18, 22] rely on appropriately translating the original piece of software, as well as the specification to be verified, to a propositional formula. The use of a SAT-Solver [10, 25, 26] then allows one to find a valuation for the propositional variables that encodes a failure. If the propositional formula is satisfiable (e.g. a valuation exists such that the formula truth value is true), then the specification is not met by the program. On the contrary, if the formula is unsatisfiable (no valuation makes the formula satisfiable), then the specification holds *within* the scope of verification provided by the user.

In the worst case, the time required to answer the satisfiability grows exponentially with respect to the number of propositional variables. Nevertheless, modern SAT-Solvers apply several heuristics leading to significant gains in analysis time.

In theory, the task of a verification tool ends when it provides a conclusive answer to the question: *is the program correct with respect to the specification?* However, the programmer daunting task of understanding the cause for such answer begins in that precise instant.

Our work assumes that verification tools may become mainstream only if proper user interfaces are provided. Essential requirements of any verification tool to become part of a programmer toolkit are their degree of automation and their ability to provide an insightful understanding of the verification result.

2. TACO A SAT-BASED BOUNDED VERIFIER

A wide variety of the tools that implement the bounded verification technique use the approach of transforming the program and its specification into a propositional formula and then passing it to a SAT-solver to check for satisfiability. TACO falls under this category.

TACO (after Translation of Annotated COde) [13] is an open source program analysis tool aimed at the verification of sequential Java programs. Given a Java program annotated with a JML [19] or a JFSL [23] specification, TACO translates both program and specification into a propositional formula. Motivated by the complexity of this process, TACO slices the involved transformations into bounded and sequential stages that perform simpler translations.

A schematic description of TACO's architecture that shows the different stages in the translation process is provided in Fig. 1. The stages presented on such diagram can be traced back unequivocally to a stage in TACO's source code.

The first transformation that TACO performs is the simplification of Java code and its corresponding specification. The objective of this stage is to normalize the kind of problems that could appear in the first translation. This transformation does not involve a translation to a different language, but only includes simplifications of Java code, the main ones being:

- Creation of default constructor if not present.
- Mapping of looping structures into a `while` form.
- Variable and parameter renaming to avoid name clashing.

Taking as input the simplified version of the program to analyze, TACO performs a translation to JDynAlloy. JDynAlloy is a relational specification language created as an intermediate representation for the translation from JML/JFSL specifications. JDynAlloy is an object-oriented language whose syntax is much simpler than other object-oriented languages such as Java or C# which allows a more compact and elegant translation to DynAlloy [12] . This translation bridges the semantic gap between an object-oriented programming language such as Java and the relational specification language DynAlloy. Given that JDynAlloy emulates structures present in OO languages, the translation to this language is almost straightforward: classes are represented as *modules*, instance variables as *fields* and methods as *programs*. However, not all Java statements have a direct correspondence in JDynAlloy. Most important considerations in this aspect are:

- As JDynAlloy has no exception management, it is simulated by means of a new variable to account for the presence of an exception.

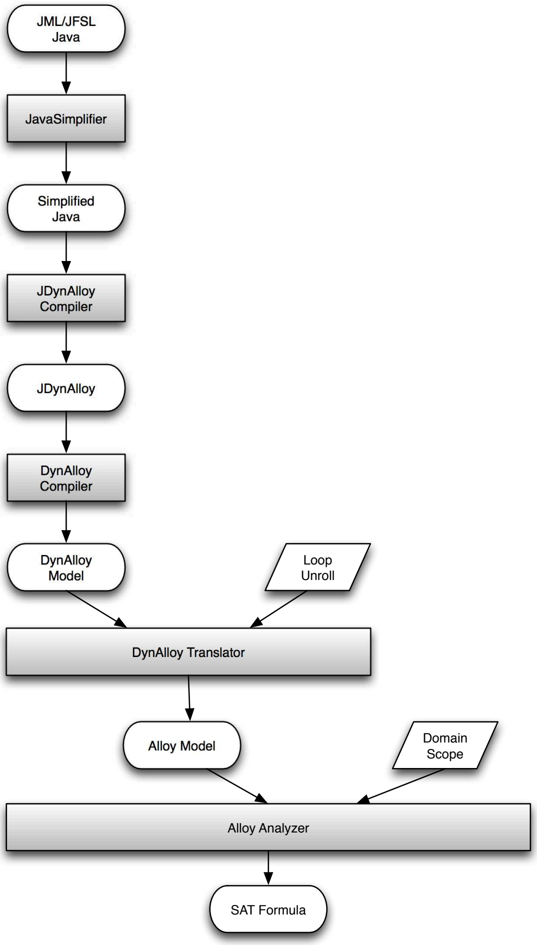


Figure 1. Schematic description of TACO's architecture

- JDynAlloy does not allow several returning points. This Java behavior is modeled introducing a new variable to acknowledge the execution of a returning statement, such as `return` or `throw`.

Once a JDynAlloy representation is obtained, the next step involves the translation to a single DynAlloy module. DynAlloy [12] is an extension of Alloy's relational logic syntax and semantics with the aim of dealing with properties of executions of operations specified in Alloy. It follows that DynAlloy extends Alloy and its relational logic. This extension provides a setting in which, besides functions describing sets of states, actions are made available, to represent state changes (i.e., to describe relations between input and output data). As opposed to the use of predicates for this purpose, actions have an input/output meaning reflected in the semantics, and can be composed to form more complex actions, using well-known constructs from imperative programming languages. The process to complete this translation is rather straightforward. The most important action of this process is the inclusion of a partial correctness assertion that states that every terminating execution of the code starting in a state satisfying the precondition and the class invariant leads to a final state that satisfies the postcondition and preserves the invariant.

DynAlloy module serves as input for the translator that performs a semantically preserving translation from the DynAlloy model to an Alloy model. Alloy [17, 16] is a formal specification language, which belongs to the class of the so-called model-oriented formal methods. Alloy is defined in terms of a simple relational semantics, its syntax includes constructs ubiquitous in object oriented notations, and it features automated analysis capabilities [15]. The Alloy language has been designed with the goal of making specifications automatically analyzable. As the aforementioned bounded verification tools, the Alloy Analyzer tool relies on off-the-shelf SAT solvers. It automatically performs an exhaustive search for counterexamples within a finite scope of analysis. In order to handle loops TACO constrains the number of iterations by performing a user-provided number of loop unrolls (LU). Therefore, the (static) analysis will only find bugs that could occur performing up to LU iterations at runtime.

Finally, the Alloy model is translated into a SAT formula. In order to build a finite propositional formula a bound is provided for each domain. This represents a restriction on the precision of the analysis. If an analysis does not find a bug, it means no bug exists within the provided scope for data domains. Bugs could be found repeating the analysis using larger scopes. Therefore, only a portion of the program domain is actually analyzed. It is worth noticing that in some programs scope and LU are not completely independent parameters. For instance, given a singly linked list with a scope of N node elements, the number of loop executions for iterating the list can not be greater then N. This dependency is a natural situation under these constraints, and similar interactions occur in other tools.

3. MOTIVATION

In order to ilustrate the reader, let us consider the source code of a Java method for extracting the minimum element from a binomial heap. Binomial heaps [5] are recursive data structures specially designed for implementing priority queues. Because of this, efficient insertion and minimum key deletion are distinguishing features of this data structure.

The JForge specification language (JFSL) is an Alloy-like language specially aimed at addressing some shortcoming of the Java Modeling Language (JML). Among its main features, JFSL allows a user to annotate a class with two class annotations: `@Invariant` which constraints the set of valid object states, and `@SpecField`. The `@SpecField` annotation is intended to raise the level of abstraction adding ghost fields to the current class declaration. In the case of binomial heap, the user can add:

- a `@Invariant` annotation for enforcing the binomial heap invariant (the `size` field stores the correct number of elements, each node has a key value less than the rest of its children, etc.) , and
- a `@SpecField` annotation declaring a ghost field named *my_nodes*. The purpose of this declaration is to predicate over the set of all reachable binomial heap node elements.

JFSL also allows the user to annotate methods with annotations about the functional behavior of the method:

- The `@Requires` annotation allows a user to capture the circumstances under which the method should be invoked.
- The `@Ensures`, `@Modifies`, `@Throws` and `@Returns` annotations are used to describe what is the expected behavior of the method (given that the preconditions are met). More specifically, the `@Modifies` annotation describe the frame condition of the method, while the `@Returns` and `@Throws` may be used to specify postconditions for normal and abnormal termination.

Figure 2 shows a code excerpt from method `extractMin()`. This source code corresponds to an implementation of binomial heaps from [21]. As the reader may notice, the program manipulates the binomial heap in a non trivial way. As a matter of fact, `extractMin()` not only accesses fields from binomial heap nodes, but also invokes methods `findMinNode` and `reverse` from the `BinomialHeapNode` class. It also calls method `unionNodes` using itself as the receiver object. The user can specify the behavior of this method by adding an `@Ensures` annotation, stating that the extraction indeed removes the minimum key from the initial binomial heap.

DOI: 10.1002/spe

```
class BinomialHeap {
 BinomialHeapNode Nodes; // header
 ...
 public BinomialHeapNode extractMin() {
  if (Nodes == null)
    return null;

    BinomialHeapNode temp = Nodes, prevTemp = null;
    BinomialHeapNode minNode = null;

    minNode = Nodes.findMinNode();
    while (temp.key != minNode.key) {
      prevTemp = temp;
      temp = temp.sibling;
    }

    if (prevTemp == null)
      Nodes = temp.sibling;
    else
      prevTemp.sibling = temp.sibling;

    temp = temp.child;
    BinomialHeapNode fakeNode = temp;
    while (temp != null) {
      temp.parent = null;
      temp = temp.sibling;
    }

    if ((Nodes != null) || (fakeNode != null)) {
      if ((Nodes == null) && (fakeNode != null)) {
        Nodes = fakeNode.reverse(null);
      } else if ((Nodes == null) || (fakeNode != null)) {
          unionNodes(fakeNode.reverse(null));
      }
    }

   return minNode;
 }
}
```

Figure 2. Excerpt from `BinomialHeap` class

Figure 3 shows the `@Ensures` annotation written in JFSL. We refer the reader to [23] for a complete description of JFSL syntax. We will describe only a subset of features of JFSL for presentation purposes. Apart from the usual logical connectives (→, &&, || etc.) and quantifiers, JFSL also provides relational operators such $@+$ and $@-$ for set union and set difference respectively, set membership (in predicate) and a function to obtain the value before method invocation ($@old$). The `@Ensures` annotation states that every execution of method `extractMin()` over a non-empty binomial heap ends with another binomial heap where:

- the return node was removed,
- the key value in the removed node is lesser or equal to any other value still stored, and
- no other keys were affected.

Once the specification is written, the user is able to try to verify the program. As we have said, the user needs to feed TACO with a scope of verification. While in some specific cases, these values may be derived from boundaries present in the business domain, in the general case the developer holds the responsibility of deciding which values she wants to check. Let us assume the user selects up to 5 iterations for each loop, a single `BinomialHeap` element and up to 13 `BinomialHeapNode` elements. Then, as previously reported in [12], TACO answers that the verification does not hold for the given scope.

As it has been mentioned, TACO uses Alloy as backend and several intermediate representations. Nevertheless, acquiring knowledge on how programs and specifications are encoded into these

```
@Ensures("
  @old(this.Nodes) ≠ null ⇒ ( my_nodes.key @+ return.key = @old(my_nodes.key))
                                  && return in @old(my_nodes)
                                  && all y : BinomialHeapNode |
                                        ( y in @old(my_nodes) ⇒ y.key ≥ return.key)
    && my_nodes=@old(my_nodes) @- return )")
```

Figure 3. An `@Ensures` annotation

languages goes far beyond the expected skills of any advanced user. In this scenario, the programmer will have to figure out by himself what prevented the verification. She will have to refer only to the program and its specification.

She may wonder if the invariant was not preserved, or which of the parts of the `@Ensures` annotation did not hold at the program exit. She may also want to inspect the initial state values, or walk through the error trace watching a given variable. In other words, she would like to inspect the result given by the verifier following a *debugging* approach. In the remaining of this article we describe our proposal for coping with these requirements.

4. THE TACO PLUG-IN

Eclipse[†] is an industrial strength, widely adopted, multi-language IDE. It can be used to develop applications in mainstream programming languages such as Java, C++ or Python and is written almost entirely in Java, just like TACO, and available for every major platform. One distinguishing characteristic of Eclipse is its support for adding new features via a sophisticated plug-in system. Because of this, writing an Eclipse plug-in appeared as a natural choice for inserting our bounded verifier user interface into an existing integrated development environment (IDE).

TacoPlug is the result of an effort to make TACO more friendly and practical to the programmer. Once installed, it allows the user to perform a bounded verification over any method of her choice. Furthermore, it provides a new Eclipse perspective (which includes different views) enabling features for debugging the program under analysis.

In the current section we show how to execute an analysis of the `extractMin()` method and we describe the different features of TacoPlug.

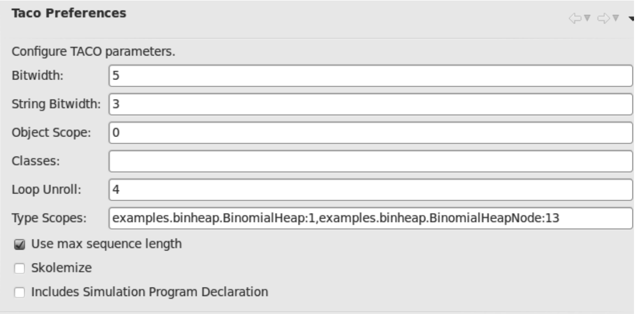
4.1. *Executing the TACO Verifier*

Executing an analysis has two important views associated. The first one, called *TACO Preferences*, can be accessed through the common Eclipse *Preferences* view on *Windows* menu and allows the user to configure all the parameters associated with an analysis. The settings are defined globally for a specific workspace.

Figure 4 shows the *TACO Preferences* view. By using this view the user can define the scope of verification. In this case, the `loopUnrollCount` parameter was set to 5 while the limits for the sizes of `BinomialHeap` and `BinomialHeapNode` domains are set to 1 and 13 respectively.

The second view includes the actual TACO launcher where the method to verify is chosen. Figure 5 shows how a verification analysis of the `extractMin()` method is launched. As with Java applications or JUnit[‡] test cases, the TACO verifier is launched through run configurations. This view allows the user to create, manage and launch a given TACO analysis.

[†] http://www.eclipse.org/
[‡] http://www.junit.org/



Taco Preferences

Configure TACO parameters.

Bitwidth: 5

String Bitwidth: 3

Object Scope: 0

Classes:

Loop Unroll: 4

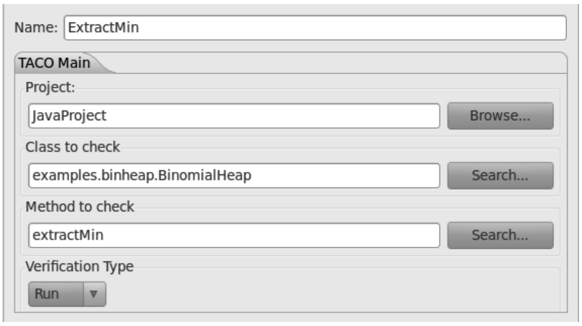
Type Scopes: examples.binheap.BinomialHeap:1,examples.binheap.BinomialHeapNode:13

☑ Use max sequence length

□ Skolemize

□ Includes Simulation Program Declaration

Figure 4. *TACO Preferences* view.



Name: ExtractMin

Project: JavaProject

Class to check: examples.binheap.BinomialHeap

Method to check: extractMin

Verification Type: Run

Figure 5. *TACO Launcher* view.

4.2. *An Eclipse Perspective for TACO*

When TACO analysis finishes the *Console* view shows the verification outcome. If an error trace was found, a popup asks the user if she wants the TACO perspective to be opened, much like when a breakpoint is hit.

This perspective, called *TACO perspective*, includes several views enabling the user to statically inspect both the method under verification and the error trace. When the perspective is shown, different views are displayed. These views allow the user to:

- inspect which part of the specification was violated,
- navigate the error trace, and
- query program values.

Further information about all these views will be given in the following subsections.

4.3. *JML/JFSL Annotation Explorer*

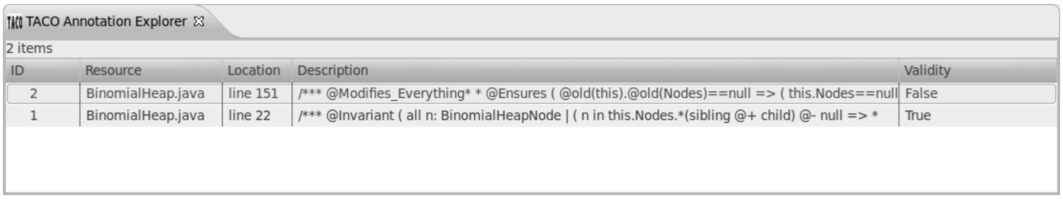
The *JML/JFSL Annotation Explorer* is the first view that becomes useful after TACO finishes. It displays a list of all annotations and their value in the error trace. By doing that, it allows the programmer to isolate which part of the JML or the JFSL specification was not met. Double-clicking on an annotation on this view will open an editor focusing the cursor on the chosen annotation.

Figure 6 shows the *JML/JFSL Annotation Explorer* for the `extractMin()` example. In this figure, the user can appreciate that the class invariant held true while the `@Ensures` annotation did not hold at program exit.

4.4. *Java Error Trace*

When debugging a program, walking through its execution is paramount. Eclipse's dynamic debugging features let the user accomplish this by using the *Stack Trace* view provided by the debugging perspective. Similarly, TacoPlug provides the ability to walk through the symbolic execution of a program, using the *Java Error Trace* view present on the TACO perspective.

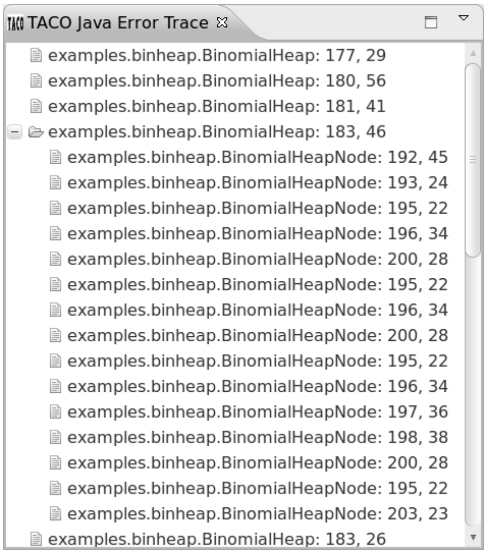
DOI: 10.1002/spe



2 items

<table>
<tr><th>ID</th><th>Resource</th><th>Location</th><th>Description</th><th>Validity</th></tr>
<tr><td>2</td><td>BinomialHeap.java</td><td>line 151</td><td>/*** @Modifies_Everything* * @Ensures (@old(this).@old(Nodes)==null => (this.Nodes==null</td><td>False</td></tr>
<tr><td>1</td><td>BinomialHeap.java</td><td>line 22</td><td>/*** @Invariant (all n: BinomialHeapNode | (n in this.Nodes.*(sibling @+ child) @- null => *</td><td>True</td></tr>
</table>

Figure 6. JML/JFSL Annotation Explorer



```
examples.binheap.BinomialHeap: 177, 29
examples.binheap.BinomialHeap: 180, 56
examples.binheap.BinomialHeap: 181, 41
examples.binheap.BinomialHeap: 183, 46
    examples.binheap.BinomialHeapNode: 192, 45
    examples.binheap.BinomialHeapNode: 193, 24
    examples.binheap.BinomialHeapNode: 195, 22
    examples.binheap.BinomialHeapNode: 196, 34
    examples.binheap.BinomialHeapNode: 200, 28
    examples.binheap.BinomialHeapNode: 195, 22
    examples.binheap.BinomialHeapNode: 196, 34
    examples.binheap.BinomialHeapNode: 200, 28
    examples.binheap.BinomialHeapNode: 195, 22
    examples.binheap.BinomialHeapNode: 196, 34
    examples.binheap.BinomialHeapNode: 197, 36
    examples.binheap.BinomialHeapNode: 198, 38
    examples.binheap.BinomialHeapNode: 200, 28
    examples.binheap.BinomialHeapNode: 195, 22
    examples.binheap.BinomialHeapNode: 203, 23
examples.binheap.BinomialHeap: 183, 26
```

Figure 7. *Java Error Trace* view

Java Error Trace view presents the error trace in a tree form where each node represents a point of the execution and parent nodes represent method calls. Clicking on any of the nodes will cause focusing the cursor on the matching line in the Java source. Stepping back and forth the error trace is easy since the user navigates the error trace without actually executing the source code.

Figure 7 presents the *Java Error Trace* view for the `BinomialHeap` example. In this particular example, the parent node represents the method call for the `findMin` and it has been expanded as an example of inner methods navigability.

4.5. *The JML/JFSL Evaluator*

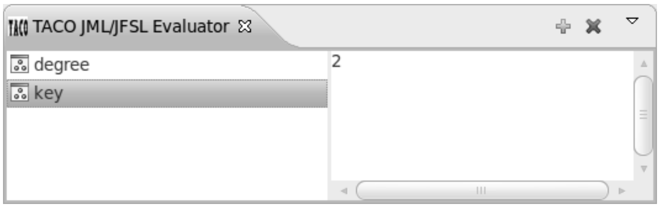
TACO, and therefore TacoPlug, is intended to be used with JML and JFSL specifications. Thus, the ability to evaluate arbitrary expressions written in these languages on different points of the error trace seemed like a natural feature to include. When a trace step is selected using *Java Error Trace* view, the user can add arbitrary JML and JFSL expressions in the *JML/JFSL Evaluator* view. Much like the *Expressions* view of Eclipse's *Debug* perspective, the *JML/JFSL Evaluator* will display their values from the current step in the error trace.

Figure 8 shows the *JML/JFSL Evaluator* including expressions *"degree"* and *"key"* corresponding to parameters `degree` and `key`.

4.6. *Java Memory Graph*

When debugging a program that includes complex and linked data structures, looking at the value of a certain variable or expression usually is not enough. In these cases, the programmer has to inspect several expressions before she can understand how objects are stored in the memory space.

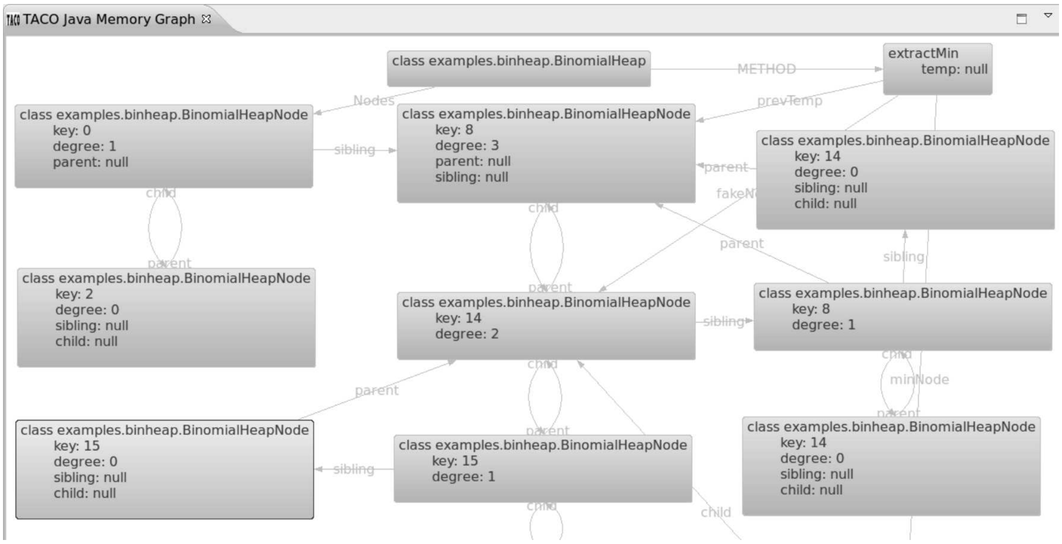
DOI: 10.1002/spe



TACO JML/JFSL Evaluator

degree	2
key	

Figure 8. *JML/JFSL Expression Evaluator* view.



TACO Java Memory Graph

```
class examples.binheap.BinomialHeap

extractMin
    temp: null

class examples.binheap.BinomialHeapNode
    key: 0
    degree: 1
    parent: null

class examples.binheap.BinomialHeapNode
    key: 8
    degree: 3
    parent: null
    sibling: null

class examples.binheap.BinomialHeapNode
    key: 14
    degree: 0
    sibling: null
    child: null

class examples.binheap.BinomialHeapNode
    key: 2
    degree: 0
    sibling: null
    child: null

class examples.binheap.BinomialHeapNode
    key: 14
    degree: 2

class examples.binheap.BinomialHeapNode
    key: 8
    degree: 1

class examples.binheap.BinomialHeapNode
    key: 15
    degree: 0
    sibling: null
    child: null

class examples.binheap.BinomialHeapNode
    key: 15
    degree: 1

class examples.binheap.BinomialHeapNode
    key: 14
    degree: 0
    sibling: null
    child: null
```

Figure 9. (Partial) *Java Memory Graph* view

The *Java Memory Graph* view displays a graph where nodes are objects and edges are field values at a given point of execution. Primitive values such as integers or booleans are shown as inner attributes of objects. If the user chooses a new step in the error trace, a new memory graph is displayed.

Figure 9 shows part of the *Java Memory Graph* view in the initial state of the execution. The complete graph matches exactly with the object diagram described in [13].

4.7. *JUnit Test Case Generation*

Once a counterexample is found, TacoPlug also provides an automatically generated (potentially partial) JUnit test case. This test case builds the offending initial heap configuration using the Java reflection mechanism. If executed, this test case is expected to exhibit the same fault found by TACO during analysis. The generation of this test case depends only upon the activation of such feature on the configuration view presented on 4.1. It is potentially partial in the sense that, if the offending behavior does not end in an unexpected exception, the user will have to write by herself the oracle for the test case.

This feature allows the user to easily collect each test case output by TacoPlug, merging them into a single test suite. Apart from regression testing purposes, another advantage of capturing a counterexample as a JUnit test case is double checking the fault against a concrete execution of the program. Skeptical users are encourage to check the fault executing the unit test case.

Figure 10 shows an excerpt of the JUnit test case that TacoPlug automatically generates for the counterexample found by TACO. Before creating the JUnit test case, the plugin checks that the visibility of the all classes found in the counterexample w.r.t. the target class's package. If all classes

DOI: 10.1002/spe

```
@Test
public void testExtractMin() {

        // Object creation
        BinomialHeap binomialHeap0 = new BinomialHeap();
        BinomialHeapNode binomialHeap0 = new BinomialHeapNode();
        BinomialHeapNode binomialHeap1 = new BinomialHeapNode();
        ...
         BinomialHeapNode binomialHeap12 = new BinomialHeapNode();
        // Field initialization binomialHeap0
        updateValue(binomialHeap0, ”Nodes”, binomialHeapNode0);
        updateValue(binomialHeap0, ”size”, 13);
        // Field initialization binomialHeapNode0
        updateValue(binomialHeapNode0, ”key”, 0);
        updateValue(binomialHeapNode0, ”degree”, 1);
        updateValue(binomialHeapNode0, ”parent”, null);
        updateValue(binomialHeapNode0, ”sibling”, binomialHeapNode1);
        updateValue(binomialHeapNode0, ”child”, binomialHeapNode2);
        ...
        // Method Invocation
        Method method = getAccessibleMethod(”BinomialHeap”, ”extractMin”, true);
        try {
            method.invoke(binomialHeap0);
        } catch (Exception e) {
            fail();
        }

}
```

Figure 10. JUnit test case automatically generated by TacoPlug

are accessible from this location, the JUnit test case is generated. Observe that visibility of fields and methods is not an issue since it is possible to change using the reflection mechanism.

4.8. *An algorithm for automatically inferring a scope of analysis*

As we previously stated, a scope of analysis is mandatory for bounded verification to work. A scope of analysis is made of: a) a number of loop unrolls to limit the length of the traces to examine, and b) a limit on the size of each object domain the code handles. Choosing a scope of analysis has a tremendous impact on the verification process, since the size and complexity of the propositional formula will depend on the setting of these parameters.

In the context of the verification of closed systems (i.e. a main method with no arguments), it is easy to over approximate the size of each object domains by unrolling the code under analysis and counting the maximum number of creation sites in any program path. Nevertheless, the task becomes more challenging when open systems (i.e. methods with complex objects as arguments, or accessing static complex objects) are our target. In order to assist the programmer in the task of systematically choosing a reasonable scope of analysis, TacoPlug comes with an automatic mechanism for (given a desired loop bound) inferring a scope for each object domain.

In Fig. 11 we show the pseudocode of our loop inference algorithm. The user must provide (apart from the desired program) a bound to the number of loop unrolls u and a bound for object domains n. The algorithm first computes the *creation_sites* storing the maximum number of creation sites for each type T in any path (feasible and non-feasiable) in the unrolled program. Then the algorithm creates a graph using the class diagram information. In this graph an edge is materialized between types T_1 and T_2 if there is a field mapping objects of T_1 to T_2. This graph is used to count the number of paths from method arguments and static data to each type T and store that value in mapping *input_scope*. Since the number of paths could be potentially infinite (i.e. a recursive field or a cycle among different types) the input argument n is used to bound this value. The final domain size is defined by adding the number of creation sites, the bounded input scope and the domain size of every subclass of type T.

```
inputs: number of loop unroll u,
        bound for object domains n,
        program P
output: a bound for each object domain
unrolled_p := unroll(P, u)
for each domain T
   creation_sites(T) := max. number of creation sites of type T
                        for any path p in unrolled_p
endfor
cdg = buildClassDiagramGraph()
for each domain T
   input_scope(T) := min(n , number of paths trasversing cdg)
endfor
for each domain T
  domain_size(T) = creation_sites(T) + input_scope(T)
  domain_size(T) += domain_size(T') T' subclass of T
endfor
```

Figure 11. Inferring domain limit algorithm

This algorithm allows the user to explore the scope of analysis by controlling only two parameters: the number of loop unrolls and the bound to the number of input objects when there is a potentially unbound heap configuration. It is easy to see that if the number of loop unrolls is kept constant, then increasing the bound on input objects will always lead to a bigger search space. The user may enable the scope inference mechanism and set the desired loop and domain bounds by setting the proper checkbox in the TACO preferences window presented in Fig. 4.

5. AN EVALUATION OF THE TACO PLUG-IN

In order to evaluate the usability of our plug-in, we will report on how useful TacoPlug was for debugging and localizing a fault in the `extractMin` method.

After launching the analysis, TACO required 73 seconds to complete its execution. Since TACO found an offending trace, the TACO perspective was opened. The first task was to inspect which of the annotations did not hold. For this task, the *JML/JFSL Annotation Explorer* was particularly useful. It showed us that the `@Invariant` annotation held, but the `@Ensures` annotation was violated by the error trace. Once we narrowed the fault location to somewhere inside the postcondition formula, we decided to display the structure of the binomial heap after executing method `extractMin`. In order to do this, we jumped to the final state of the error trace using the *Java Error Trace* view and, selecting the *Java Memory Graph* view, we were able to visualize the final state of the program. At this point, we needed to exhaustively evaluate each `@Ensures` sub-formula. As stated on section 3, the `@Ensures` formula consist of three sub-formulas:

1. the return node was removed,
2. the key value in the removed node is less or equal to any other value still stored, and
3. no other keys were affected.

Checking these conditions against the final state of the program, we were able to discover that the first two sub-formulas held. Therefore, the fault should be referred to the condition specified in the third sub-formula, which can be written in JFSL as follows:

```
my_nodes=@old(my_nodes) @- return
```

Counting the number of nodes in the binomial heap on the final state of execution, we noticed that the binomial heap was composed of 10 nodes. Contrasting this value against the initial state of execution was necessary. By clicking on the first step of the error trace a new memory graph was displayed. We discovered that the initial binomial heap had 13 nodes. This meant that the expected number of nodes after `extractMin` was 12. This led us to conclude that 2 nodes were missing at

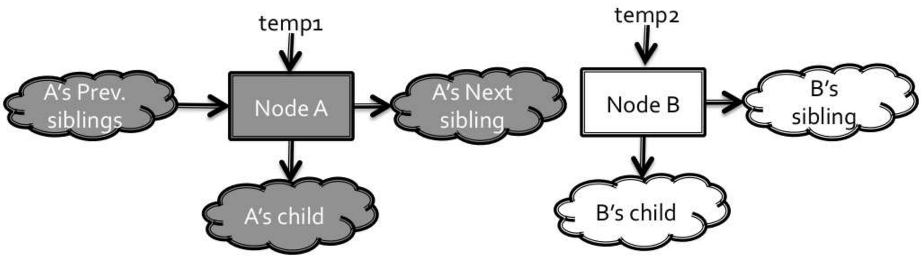


Figure 12. heap configuration before merging two different binomial heaps

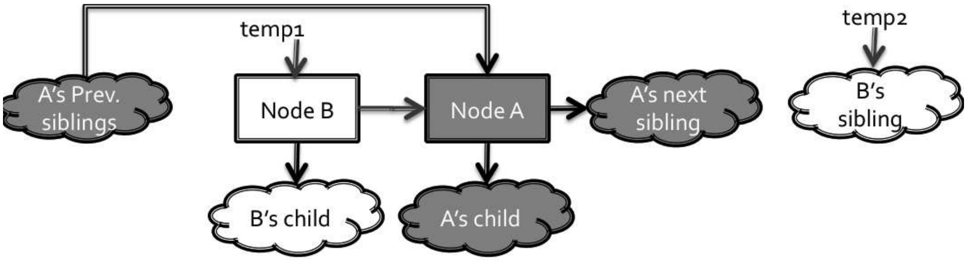


Figure 13. heap configuration after merging when degree(temp1)>degree(temp2)

the final state. In other words, the amount of nodes after executing method `extractMin` was not correct.

Once the faulty behavior was instantiated for this error trace, the next step was to find where the fault was located. For this, two views were crucial: *Java Error Trace* and *Java Memory Graph*. We asked the following question: *What is the point in the error trace where the number of nodes reaches* 10 *for the first time?* In order to answer this question, we performed a binary search in the Java error trace, checking on each step the number of nodes stored in the binomial heap. By doing that, we were able to isolate the misbehavior to the helper method `merge`. Apparently, merging two binomial heaps of **7** and **5** nodes resulted in a new binomial heap of only 10 nodes.

After determining that some anomalous behavior was contained in method `merge`, we inspected how these two binomial heaps were merged. This method merges two lists of nodes observing the value of field `degree`. After inspecting the error trace going back and forth, we were able to

- understand the intended loop invariant for merging two binomial heaps, and more importantly,
- localize the fault in source code.

The fault consisted in a mishandling in how references from nodes already merged were being updated. More specifically, when merging $x_0, \dots, x_n$ and $y_0, \dots, y_m$, if y_0 is inserted before x_0, then references of other nodes to x_0 should be updated to y_0. In Fig. 12 we show the heap configuration before merging two binomial heaps (references *temp1* and *temp2*). After the execution of the merge operation (Fig. 13) we can see that node A previous siblings are no longer reachable from the current binomial heap.

Finally, in order to double check the error using a dynamic environment, we executed the generated JUnit test case and proved that, when executing the `extractMin` method with that particular input, some binomial heaps nodes were lost. Furthermore, we were able to trace back the fault to the same execution point found using the TacoPlug approach, making things consistent as expected.

6. PLUG-IN DETAILS

The design and development of the plug-in consisted on a two phase project, both of which presented several challenges. While the second phase of the project consisted on the integration of TACO to Eclipse by means of a plug-in, the first phase involved the transformation of the batch processing tool TACO to a interactive program to provide support for the intended functionalities of the plug-in. These two phases, the main techniques and challenges involved will be described in this section.

The first phase involved enhancing TACO in order to provide functional support for all the features we intended TacoPlug to support. To begin with, this meant the transformation of TACO from a tool that process an input (annotated source code) and returns an output (in SAT cases, a counterexample) to a tool in which the output can be queried and analyzed. The counterexample returned by TACO until the beginning of this project was presented in Alloy language. As the intention was to abstract the developer from the TACO implementation and the intermediate languages it uses, a mechanism to translate that counterexample to a Java-like language was needed.

As it has been mentioned before, TACO's architecture consists of a pipeline that transforms the original annotated program source code to a propositional formula. Thus, designing and developing a new stage to build a Java counterexample seemed as the natural choice. The main responsibilities of this stage include:

- building a Java trace for the fault,
- checking whether class invariant and method postcondition held in the final state of execution.

For both of this tasks a mechanism presented on [27] was used. This mechanism, when applied to an Alloy counterexample, returned a DynAlloy counterexample, which brought us a little bit closer to the abstraction level desired. This counterexample includes a DynAlloy trace for the fault and an entry point to evaluate DynAlloy expressions on each step of the trace.

Using the DynAlloy trace in combination with a reference to the original Java statements while the analyzed program traverses the TACO pipeline, provided the necessary tools to build a Java trace for the fault.

In order to check the validity of the class invariant and method postcondition in the final state of execution, a technique that provided the means to evaluate JML/JFSL expressions was required. The main challenge in designing such technique was that in TACO's pipeline, variable renaming is done. Therefore, to be able to evaluate JML/JFSL expressions, the exact same variable renaming needed to be performed in such expressions, which meant saving the original renaming dictionary in a new object we called *TacoContext*. Saving the variable renaming dictionary for the pipeline stages that included such mechanism in the TacoContext, provided the means to do the variable renaming in the class invariant and method postcondition. Replacing the corresponding variable names in either formula and using the mechanisms in TACO to translate a JML/JFSL formula to a DynAlloy formula was all that we needed in order to evaluate them in the context of the DynAlloy counterexample.

Being able to build a Java trace of the detected fault and test whether the class invariant and method postcondition held in the final state of execution are the back-ends for the *Java Error Trace* and *JML/JFSL Annotation Explorer*.

Furthermore, the described technique is general enough to grant the user with the possibility of querying the DynAlloy counterexample using JML/JFSL expressions, a feature needed to build the *JML/JFLS Evaluator*. However, as the returning type of evaluating invariant and postcondition formulas is known to be `Bool`, a way of reconstructing a Java-like-return-type object was not developed. This enhancement was incorporated in a second stage and a full description of such is not included here because of space restrictions. The construction of the Java-like return object is based on the obtainment of its Java type, using properties present on the Alloy object returned upon evaluation of DynAlloy expressions; and the computation of the value of its fields, which means building and evaluating DynAlloy expressions for each of them. In this process, the aliasing problem was also addressed.

Finally, the development of a technique to construct the state of the heap at a specific point of execution is based on an analysis of the scoped variables and the evaluation mechanism described

DOI: 10.1002/spe

before. The construction of the graph representing this state provides the basis for the *Java Memory Graph* view.

These functionalities were a requirement to start writing the plug-in itself. Adapting TACO added around 1.5K LOC to the TACO project. This made TACO grow to 30K LOC, without counting any depending project.

The second phase included the actual construction of the plug-in. During its development, we worked mainly on usability issues. This phase included not only the development of the components discussed on section 4, but also of some computations that occur transparently to the developer saving her of further configuration in the use of TACO. Both user interactive components and hidden computations development will now be presented.

TacoPlug development, as every Eclipse plug-in, bases its architecture on Eclipse's PDE[§] infrastructure. PDE includes the necessary tools for the design, development, testing and deployment of a plug-in. Components of the plug-in are defined as *extensions* for *extension points* provided by PDE. Each of the components exhibited as part of TacoPlug consists of one extension. Furthermore, some of the extensions included the use of third party plug-ins, such as Zest[¶] which provided the graph drawing functionality for the *Java Memory Graph* view.

Moreover, a type of editor, called TACO Editor, was included giving the user the possibility of viewing different intermediate representation languages that TACO uses for analysis purposes. This is not reported on section 4 because, as we have already stated, it will not be useful to the majority of plug-in users, but only to a small group of advanced users.

For the reader interested on plug-in development, an Eclipse plug-in called PDE Incubator Spy[‖], could play a major role in helping to understand the Eclipse plug-in architecture. Incubator Spy aims to provide a simple tool to introspect Eclipse in terms of what a plug-in developer would find useful.

Another thing worth mentioning is that, in contrast to the command line interface of TACO, some parameters are automatically inferred when using the plug-in. For example, TACO requires that the user provides the set of relevant classes for the verification. Using JDT Project[**] we were able to infer the set of relevant classes for the method under verification and save the user the necessity of configuring them. Another example would be the inclusion of a compiled version of these relevant classes in the *class loader*, without which returning Java-like objects as the result of the evaluation of JML/JFSL expressions would be impossible.

7. RELATED WORK

Apart from TACO, other SAT-based verification tools are CBMC [3], Saturn [22], and F-Soft [14] for the analysis of C code, and Miniatur [9] and JForge [6] for Java code analysis. Unfortunately, we were not able to evaluate the usability of Miniatur and F-Soft since both tools are not publicly available. Although CBMC pinpoints the violated assertion and prints out an error trace, it does not allow the user to dynamically inspect the error trace, nor inspect expressions along the error trace. The current JForge distribution includes an Eclipse plug-in. As with CBMC, the JForge plug-in highlights the violated parts of the specification. It also presents memory graph visualizations, but restricted to the entry and exit states of the error trace. The JForge plug-in allows the user to visualize the offending program trace, but described in a intermediate representation language.

In [20], authors generate an executable C# program that reproduces an error found by the Spec# verifier. The purpose of this generated program is not only understand the failure, but also to detect spurious errors. Unlike Taco, Spec# follows a modular approach for verification. In the generated program, loops and invocations to other programs are replaced with the program specification.

[§] http://www.eclipse.org/pde/
[¶] http://www.eclipse.org/gef/zest/
[‖] http://www.eclipse.org/pde/incubator/spy/
[**] http://www.eclipse.org/jdt/

DOI: 10.1002/spe

This mimics Spec#'s modular semantics. Since the generated program is actually executed, moving backwards in the error trace is not allowed.

To the best of our knowledge, the Boogie Verification Debugger (BVD for short) is the closest work to our approach. It features navigating an error trace forward and backward, and inspecting several variables and access paths. Visual Studio plug-ins are presented for the verification front-ends VCC and Dafny. Both tools perform modular verification of programs (specifications of helper methods and loops invariants have to be provided by the user). Although it features some interesting listings such as showing the programmer all aliases to a given object, BVD offers no graph visualization and no test input generation out of the error model produced by the decision engine.

8. CONCLUSIONS AND FUTURE WORK

In this work we presented TacoPlug, an Eclipse plug-in that provides a proper user interface for a verification tool, TACO. TacoPlug provides the developer the possibility of performing a deeper analysis of the TACO result which helps in the task proposed in the introduction of this article: finding and understanding the cause of the fault.

The means by which the plug-in enables the developer to perform this analysis include: (a) a JML/JFSL annotation explorer, allowing the user to narrow the fault to a specific formula of the specification; (b) an error trace explorer, which permits the user the navigation of the instructions that result in the fault; (c) an expression evaluator, providing the possibility of evaluating JML/JFSL expressions in any point of the trace; (d) a graph visualisation of the heap in any point of the trace, providing a clearer picture of how memory is allocated in any point of the trace. Additionally, TacoPlug provides the possibility of generating a unit test case, which can be run using JUnit testing framework, that exposes the same bug than the counterexample found by TACO. This is especially useful for development processes which include unit testing development, such as TDD (Test Driven Development), and execution as a core stage. With the inclusion of this test case generation as a part of TacoPlug, the developer now possesses a test case available to be run as part of a regression test. Finally, keeping in mind that verification tools will only go mainstream if a 'push-button' nature is provided, TacoPlug provides the developer an heuristic mechanism to automatically calculate the scope of the analysis she wants to perform.

This extension has proven to be useful for execution of a bounded verification and the posterior analysis of the outcome, facilitating the user the debugging and localizing of a fault. For example, it has been helpful in elucidating why the `extractMin` method was not correct. We believe that tools like TacoPlug are necessary to move verification into the hands of a wider range of users.

Both TACO and TacoPlug source code is publicly available for download at `http://www.dc.uba.ar/taco`.

For future work, we plan to conduct user experiments with verification of a wide range of programmers, which might include students. From a technical point of view, improving the efficiency of information recovery and predicate evaluation algorithms is one of the main objectives of a future version of the plug-in.

There is extensive work on increasing the verification capabilities of TACO [2]. More recently, a framework for test input generation using the TACO infrastructure was proposed [1]. We plan to extend our current plugin to allow the user automatically create a test suite given a chosen criteria.

The current JUnit generated by TACO relies on the Java reflection mechanism for appropriately building the offending heap configurations. We understand that it is much more convenient to a user obtaining the same heap configuration by invoking a sequence built of the accessible class methods. In particular, this rules out those spurious heap configurations that are not feasible under the current set of constructors and other class methods. In order to do so, we plan to investigate how to apply genetic algorithms to guide the search of the desired heap configuration. More specifically, we plan to extend the EvoSuite tool [11] by adding a custom fitness function which tailors the search towards building the offending heap configuration.

From a user perspective, we would like to add a *diff* visualization tool that allows the user to isolate the differences between two memory states [24]. We are also working in providing automatic support for improving failure understanding by slicing away those statements that are not directly related to the violated specification. Other ideas that we have in mind include: JDynAlloy and DynAlloy syntax highlighting; an analysis administrator, enabling the user to perform more than one analysis at a time; partial formula evaluation, allowing the user to select a sub-formula from the specification and automatically analyse it.

ACKNOWLEDGEMENT

Alessandra Gorla gave helpful comments on earlier revisions of this paper. This work has been supported by the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS). Contract/grant sponsor: UBACyT-20020110200075, CONICET-PIP 11220110100596CO.

REFERENCES

1. Abad et. al., *Tight Bounds + Incremental SAT = Better Test Generation under Rich Contracts*, submitted for publication to ICST 2013, International Conference on Software Testing 2013.
2. Cuervo Parrino B., Galeotti J.P., Garbervetsky D., and Frias M., *A dataflow analysis to improve SAT-based program verification*. In SEFM 2011: Software Engineering and Formal Methods, page 138–153 - 2011
3. Clarke, E., Kroening, D., Lerda F. *A Tool for Checking ANSI-C Programs*. In TACAS 2004. LNCS (2988) pp. 168–176.
4. Clarke, E., Kroening, D., Sharygina, N., Yorav, K. *SATABS: SAT-based Predicate Abstraction for ANSI-C*. In TACAS 2005. LNCS (3440) pp. 570-574.
5. Cormen, T., Leiserson C., Rivest R., and Stein C., *Introduction to Algorithms (3. ed.)* MIT Press, 2009.
6. Dennis G. *A Relational Framework for Bounded Program Verification*. MIT PhD Thesis. July 2009.
7. Dennis, G., Yessenov, K., Jackson D., *Bounded Verication of Voting Software.* in VSTTE 2008. Toronto, Canada, October 2008.
8. Dennis, G., Chang, F., Jackson, D. *Modular Verication of Code with SAT*. ISSTA06, pp. 109-120, 2006
9. Dolby J., Vaziri M., Tip F., *Finding Bugs Efficiently with a SAT Solver*, in ESEC/FSE'07, pp. 195–204, ACM Press, 2007.
10. Een, N., Sorensson, N., An extensible SAT-solver. In LNCS. Volume 2919 (2004) pages 502–518.
11. Fraser G. and Arcuri A., *EvoSuite: automatic test suite generation for object-oriented software*, in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, New York, NY, USA, 2011, pp. 416-419.
12. Frias, M. F., Galeotti, J. P., Lopez Pombo, C. G., Aguirre, N., *DynAlloy: Upgrading Alloy with Actions*, in ICSE'05, pp. 442–450, 2005.
13. Galeotti J., Rosner N., Lopez Pombo, C., Frias M., *Analysis of Invariants for Efficient Bounded Verification*, in ISSTA 2010, Trento, Italy.
14. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P., *F-Soft: Software Verification Platform*. In CAV'05, pp. 301–306, 2005.
15. Jackson, D., *A micromodels of software: Lightweight modelling and analysis with Alloy* MIT Laboratory for Computer Science, Cambridge, MA, 2002.
16. Jackson, D., *Software Abstractions.* MIT Press, 2006.
17. Jackson, D., *Alloy: a lightweight object modelling notation* ACM Transactions on Software Engineering and Methodology, 2002.
18. Jackson, D., Vaziri, M., *Finding bugs with a constraint solver* in ISSTA00, pp. 14–25, 2000.
19. Leavens, G., Baker A., Ruby C. *Preliminary design of JML: a behavioural interface specification language for Java*. ACM Software Engineering Notes. Volume 31 Issue 3, May 2006.
20. Müller P., and Ruskiewicz J., *Using Debuggers to Understand Failed Verification Attempts*. In Formal Methods (FM) 2011, pp. 73–87, 2011.
21. Visser W., Păsăreanu C. S., Pelánek R., *Test Input Generation for Java Containers using State Matching*, in ISSTA 2006, pp. 37–48, 2006.
22. Xie, Y., Aiken, A., *Saturn: A scalable framework for error detection using Boolean satisfiability*. in ACM TOPLAS, 29(3): (2007).
23. Yessenov K. *A Light-weight Specification Language for Bounded Program Verification*. MIT MEng Thesis. May 2009.
24. Zimmermann T., Zeller A. *Visualizing Memory Graphs*. In Software Visualization 2001, pp. 191–204.
25. Goldberg E., Novikov Y. *BerkMin: A fast and robust SAT-solver*. In Proceedings of the conference on Design, automation and test in Europe, pp. 142–149. IEEE Computer Society.
26. Moskewicz, M., Madigan, C., Zhao Y., Zhang L., and Malik S. *Chaff: engineering an efcient SAT solver*. In J. Rabaey editor, Proceedings of the 38th conference on Design automation, pages 530535, Las Vegas, Nevada, United States, 2001. ACM Press.
27. P. Bendersky *Hacia un entorno integrado para la verificación de contratos utilizando SAT Solvers.* Masters thesis, Universidad de Buenos Aires, 2010.

DOI: 10.1002/spe