TacoPlug: An Eclipse Plug-In for TACO

Marcos Chicote∗ and Juan Pablo Galeotti∗†
∗Departamento de Computación, FCEyN, UBA
Buenos Aires, Argentina
{mchicote,jgaleotti}@dc.uba.ar
†CONICET, Argentina

Abstract—In this work we present TacoPlug, an Eclipse plug-in that lets users explore error traces output by the bounded verifier TACO. TacoPlug 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. We show the usability of our tool by means of a motivational example taken from a real-life software error.

Keywords—Static analysis, bounded verification, Eclipse plug-in, TACO.

I. 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., this expansion comes with an undesirable companion: 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 artifacts 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 failure.

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 allow the programmer an insightful understanding of the verification result.

TACO [7] is a SAT-based tool specially aimed at the verification of sequential Java programs. Given a Java program annotated with a JML [10] or a JFSL [14] specification, TACO translates both program and specification into a propositional formula. In order to build this propositional formula, the user should provide:

- limits for the sizes of the object domains, and
- a limit for the number of loop iterations.

We will refer to these limits as the scope of the bounded verification. 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 given by the user. In order to determine if a formula is satisfiable or not, TACO feeds the formula into a SAT-Solver [5]. 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.

The rest of this article is organized as follows. Section II describes the purpose of our plug-in. Section III describes the features and usage of TacoPlug on the motivational example, and provides several screenshots of our tool. Section IV evaluates the usability of our plug-in in debugging a previously known failure. Section V describes the implementation and the challenges we faced during the development of the plug-in. Section VI presents related work, finally Section VII concludes and introduces our planned future work.

II. MOTIVATION

In order to illustrate the reader, let us consider the source code of a Java method for extracting the minimum element from a binomial heap. Binomial heaps [2] 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. Figure 1 shows a code excerpt from method extractMin(). This source code corresponds to a
implementation of binomial heaps from [12]. As the reader may notice, the program manipulates the binomial heap in a non trivial way. The user also specifies the binomial heap behavior by adding an @Invariant annotation and an @Ensures annotation. The former constrains the set of valid object states, while the latter states that the extraction indeed removes the minimum key from the initial binomial heap.

JForge constructs allows the user to write first order logic formulas with relational operators such as navigation, union, disjunction, transitive relational closure, etc. Figure II shows the @Ensures annotation written in JFSL. A more detailed description of the JForge specification language may be found in [14]. 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. 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 [6], TACO answers that the verification does not hold for the given scope.

TACO uses Alloy [9] and DynAlloy [6] as backends. The intermediate representations and the analysis results for each of these tools are output by TACO. Nevertheless, acquiring knowledge on how programs and specifications are encoded into these languages go far beyond the 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.

III. THE TACO PLUG-IN

Eclipse\(^1\) 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.

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 his 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.

A. Executing the TACO Verifier

Executing an analysis has two important views associated. The first one, called TACO Preferences, can be accessed

\(^1\)http://www.eclipse.org/
through the common Eclipse Preferences view on Windows menu and allows the user to configure all the parameters associated to an analysis. The settings are defined globally for a specific workspace.

Figure 3 shows the TACO Preferences view. By using this view the user can define the scope of verification. In this case, the loopUnrollCount parameter was setted to 5 while the limits for the sizes of BinomialHeap and BinomialHeapNode domains are setted to 1 and 13 respectively.

Figure 3. TACO Preferences view.

The second view includes the actual TACO launcher where the method to verify is chosen. Figure 4 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.

Figure 4. TACO Launcher view.

B. 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 hitted.

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.

C. JML/JFSL Annotation Explorer

The first view that becomes useful after TACO finishes is the JML/JFSL Annotation Explorer. 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 and editor focusing the cursor on the chosen annotation.

Figure 5 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.

D. 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.

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 6 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.

E. 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 7 shows the JML/JFSL Evaluator including expressions "degree" and "key" corresponding to parameters degree and key.
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. 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 8 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 [7].

IV. AN EVALUATION OF THE TACO PERSPECTIVE

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. We exhaustively evaluate each @Ensures subformula at the final state. Then, we discovered that only the following subformula did not hold:

\[
\text{my\_nodes} = \text{old(my\_nodes)} \ (- \text{return})
\]

In other words, the amount of nodes after executing method extractMin was not correct. Once we understood what went wrong with respect to the specification, 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. Then, by selecting the Java Memory Graph view, we counted 10 nodes in the binomial heap. By clicking on the first step of the error trace a new memory graph was displayed. We discovered that the initial binomial heap has 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 the final state.

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 \( x_0, \ldots, x_n \) and \( y_0, \ldots, y_m \), if \( y_0 \) is inserted before \( x_0 \), then references of other nodes to \( x_0 \) should be updated to \( y_0 \).

V. PUG-IN DETAILS

The design and development of the plug-in consisted on a two phase project.

The first phase involved adapting TACO in order to provide functional support for all the features we intended TacoPlug to support. This adaptation consisted basically in:

- tracing Java and DynAlloy statements,
- evaluating of JML and JFSL expressions.

DynAlloy already provided the way to evaluate DynAlloy expressions in different steps of the DynAlloy trace. By tracing Java statements to DynAlloy statements we were able to construct an error trace at the Java level. Secondly, by mapping JFSL and JML expressions to their DynAlloy counterparts, we were able to evaluate these expressions in a particular point of the Java error trace.

These functionalities where 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 the development of the components mentioned on section III. Eclipse’s PDE\(^2\) infrastructure was used and Zest\(^3\) plug-in 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 III 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. 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\(^4\) we were able to infer the set of relevant classes for the method under verification and save the user the necessity of configuring them. TacoPlug consists of 2500 LOC.

VI. RELATED WORK

Apart from TACO, other SAT-based verification tools are CBMC [1], Saturn [13], and F-Soft [8] for the analysis of C

---

\(^2\)http://www.eclipse.org/pde/
\(^3\)http://www.eclipse.org/gef/zest/
\(^4\)http://www.eclipse.org/jdt/
code, and Miniatur [4] and JForge [3] 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 [11], 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 using the program specification. 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.

VII. 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. 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 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. 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 and TACO’s ability to generate jUnit test cases. 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.

Finally, we strongly believe functionalities to introspect Eclipse (such as PDE Incubator Spy6) could play a major role in helping developers to understand the Eclipse plug-in architecture.

REFERENCES


5http://www.junit.org/
6http://www.eclipse.org/pde/incubator/spy/