Algorithmic Debugging of Java Programs

R. Caballero[1,2]

Facultad de Informática
Universidad Complutense de Madrid
Madrid, Spain

C. Hermanns [3]

Institut für Wirtschaftsinformatik
Universität Münster
Münster, Germany

H. Kuchen[1,4]

Institut für Wirtschaftsinformatik
Universität Münster
Münster, Germany

Abstract

In this paper we propose applying the ideas of declarative debugging to the object-oriented language Java as an alternative to traditional trace debuggers used in imperative languages. The declarative debugger builds a suitable computation tree containing information about method invocations occurred during a wrong computation. The tree is then navigated, asking the user questions in order to compare the intended semantics of each method with its actual behavior until a wrong method is found out. The technique has been implemented in an available prototype. We comment the several new issues that arise when using this debugging technique, traditionally applied to declarative languages, to a completely different paradigm and propose several possible improvements and lines of future work.

Keywords: Declarative Debugging, Object-Oriented Languages.

1 Introduction

Nowadays the concept of *encapsulation* has become a key idea of the software development process. Applications are seen as the assembly of different encapsulated software components, where each component can in turn be composed of other simpler components. The encapsulation here means that the programmer only needs to know *what* each component does (the semantics), without worrying about *how*

[1] This author has been funded by the projects TIN2005-09207-C03-03 and S-0505/TIC/0407.

[2] Email: rafa@sip.ucm.es

[3] Email: hermannc@wi.uni-muenster.de

[4] Email: kuchen@uni-muenster.de

this is done (the implementation). Object oriented languages such as Java [16] are based on this concept, allowing the definition of flexible, extensible, and reusable software components, thus saving time and avoiding errors in the final program.

While this view of the software as an assembly of components has been successful and it is widely used during the phases of design and implementation, it has had little influence on later phases of the software development cycle such as testing and debugging. Indeed, the debuggers usually included in the modern IDEs of languages such as Java, are sophisticated *tracers*, and they do not take advantage of the component relationships.

Declarative debugging, also known as *algorithmic debugging*, was introduced by E. Y. Shapiro in [14] as an alternative to trace debuggers for the Logic Programming paradigm, where the complex execution mechanism makes traditional debugging less suitable. The idea was afterwards employed in other declarative programming paradigms such as functional [11] and functional-logic [3] programming.

A declarative debugger starts when the user finds out some unexpected behavior while testing a program, i.e. an initial symptom. Then the debugger builds a tree corresponding to the computation that produced the initial symptom. Each node of this tree corresponds to the result of some subcomputation, and has a fragment of program code associated, namely the fragment of code needed for producing the result. The children of a node correspond to the results of those subcomputations that were necessary to obtain the parent result. In particular the root of the tree corresponds to the result of the main computation. The user navigates the tree looking for a node containing a non-valid result but whose children produced valid results. Such a node is considered a *buggy node* and its associated fragment of code is pointed out as erroneous because it has produced a wrong output result from the correct input results of its children.

In this paper we apply the idea of declarative debugging to the object oriented language Java. Starting from some erroneous computation, our debugger locates a *wrong method* in the debugged program. The task of the user during a debugging session is reduced to checking the validity of the results produced by some method calls occurring during the computation. Thus, our tool abstracts away the details of the implementation of each method, deducing the wrong method from the intended semantics of the methods and the structure of the program.

In order to improve the efficiency of the tool we propose using a test-case generator such as GlassTT [8,9,10]. This tool divides the possible input values of each method into equivalence classes using some coverage criteria. The correct/incorrect behavior of a method call for any representative of an equivalence class will entail the validity/non-validity of the method for the other members of the class. Therefore the debugger can use this information in order to infer the state of several nodes from a single user answer. Although still in the early stages of study, we think that this improvement can dramatically reduce the number of nodes considered during a debugging session.

The idea of using declarative debugging out of declarative programming is not new. In 1989 N. Shahmehri and P. Fritzson presented in [13] a first proposal, further developed by the same authors in [5]. In this works a declarative debugger for the imperative language *Pascal* was presented. The main drawback of the proposal was

that the computation tree was obtained by using a program transformation, which limited the efficiency of the debugger. The differences of our approach w.r.t. these earlier proposals are:

- Java is a language much more complex than Pascal. The declarative debugging of programs including objects and therefore object states introduces new difficulties that had not been studied up to now and that we tackle in this paper.
- Modern languages such as Java offer new possibilities for implementing the declarative debugger. In our case we have based the implementation of our prototype on the *Java Platform Debugging Architecture* (JPDA) [7]. JPDA has an event-based architecture and uses the method-entry and method-exit events in order to produce the computation tree. The result is a much more efficient tool.

A more recent related work [6] presents a declarative debugger for Java that keeps information about most of the relevant events occurred during a Java computation, storing them in a deductive database. The database can be queried afterwards by the user in order to know the state of the program (variables, threads, etc.) in different moments of the execution in order to infer where the bug is located. The main difference between both proposals is that our debugger concentrates only on the logic of method calls, storing them in a structured way (the computation tree) which allows the debugger to deduce the wrong method from the user answers.

In the next section we present the application of the ideas of declarative debugging to the case of Java programs. Section 3 introduces the idea of using a test-case generator in order to reduce the number of questions that the user must answer before finding the bug. In Section 4, we discuss the limitations of our prototype. Finally the work ends presenting some conclusions and future work.

2 Declarative Debugging

In this section we present the ideas of declarative debugging applied to the object-oriented language Java, and its prototype implementation.

2.1 Computation Trees

We start by defining the structure of the computation trees used by our debugger. Since the aim of the tool is detecting wrong methods in Java programs, each node of the tree will contain information about some method call occurred during the computation being analyzed. Let N be a node in the computation tree containing the information of a method call for some method f. Then the children nodes of N will correspond to the method calls occurring in the definition of f that have been executed during the computation of the result stored at N. For example, consider a method f defined as:

```
public int f(int a) {    if (a>0) return g(a); else return h(a); }
```

Then any node associated to a method call for f will have exactly one child node, that will correspond to either a method call to g (if the parameter is positive) or to h (otherwise). Thus, as we will see in the example of the next subsection, a method call occurring inside a loop statement can produce several children nodes.