The Journal of Logic and Algebraic Programming 81 (2012) 390–407

Contents lists available at SciVerse ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Automated debugging based on a constraint model of the program and a test case ☆

Franz Wotawa *, Mihai Nica, Iulia Moraru

Technische Universität Graz, Institute for Software Technology, Inffeldgasse 16b/2, A-8010 Graz, Austria

ARTICLE INFO

Article history:
Available online 15 March 2012

Keywords:
Automated debugging
Algorithmic debugging
Model-based debugging
Constraints
Constraint satisfaction problem

ABSTRACT

Debugging, i.e., fault localization, in case of a detected failure is a time consuming and intricate task. The automation or at least partial automation of debugging is therefore highly desired. In this paper, we discuss some of the most recent approaches for debugging namely spectrum-based, slicing-based, and model-based debugging. We focus on the latter, and introduce the underlying theory as well as discuss empirical results obtained from our implementation. The model-based approach we present in this paper relies on a constraint representation of a program that is equivalent to the original program in terms of the input-output behavior under some reasonable assumptions. By using constraints for representing programs and subsequently test cases we are able to state the debugging problem as a constraint satisfaction problem that can be effectively solved using a todays constraint solver. The given empirical results indicate that the approach can be used for debugging smaller programs in less than 1 s. Moreover, we briefly compare the three approaches and suggest a combination of them in order to improve the results and the overall necessary running time.

1. Introduction

Program debugging, i.e., the detection, localization, and correction of programs, is generally considered a hard problem especially after program deployment, but of huge practical value. Support of program debugging helps to keep direct and indirect costs of software development low. Faults that are corrected early in the development process cause less costs than faults revealed after shipping the software to the customers. Therefore, most of the research activities since the beginnings of software engineering have focused on verification and validation in order to ensure program correctness. Only a little research effort has been spent in developing tools for debugging. In this paper, we discuss some, but not all methods for automated fault localization and present one, i.e., model-based debugging, in more detail.

In the context of the paper, program debugging is defined as the activity carried out by humans or a program itself that localizes a root cause in the source code of a program, which is responsible for an observed behavior deviating from the expected one. Obviously, after finding the root cause, we are also interested in making the corrective changes, but this part of debugging as a whole is not in the focus of this paper. The given definition of debugging is a very general one. Until now, we have not introduced any restriction regarding how to observe a deviation. For example, such a deviation might come from user demands that are not implemented in the deployed program. Such a root cause requires adding functionality to the program and has to do with re-design. Another reason for inconsistencies is that the program does not pass all test cases. As a consequence, verification reveals a faulty behavior and the cause usually has to be tracked down to parts of the source code responsible for the misbehavior. Note that a program might fail passing a test case because the program computes a wrong value for a variable or raises an exception like a division-by-zero exception.

☆ Authors are listed in revers alphabetical order.

* Corresponding author.
E-mail addresses: wotawa@ist.tugraz.at (F. Wotawa), mnica@ist.tugraz.at (M. Nica), imoraru@ist.tugraz.at (I. Moraru).

1567-8326/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jlap.2012.03.002

```
 1.        i = 1;
 2.        min = input[0];
 3.        max = input[0];
 4.        while (i < length) {
 5.             if (input[i] < min) {
 6.                  min = input[i];
 7.             }
 8.             if (input[i] > max) {
 9.                   max = input[i];
10.             }
11.             i = i + 1;
12.        }
13.        result = min * max;   // BUG should be result = min + max;
```

Fig. 1. A program fragment computing the minimum, maximum, and sum of both for an array of integers.

Table 1
The test suite used to verify the program from Fig. 1.

Test case	Input / input	Expected output
A	[1]	result = 2, min = 1, max = 1
B	[1, 2]	result = 3, min = 1, max = 2
C	[2, 1, 3, 0]	result = 3, min = 0, max = 3
D	[0, 1, 2, 3]	result = 3, min = 0, max = 3
E	[2, 1]	result = 3, min = 1, max = 2

In the approach presented in this paper we rely on some restrictions. We assume that the source code of the program as well as a test suite comprising at least one failing test case is given. We further restrict debugging to cases where the original program computes a wrong value for at least one program variable. We assume that the program to be debugged is syntactically correct, does not comprise any type errors and infinite loops, and that the corrected program is a close variant of the original one.

For the purpose of explaining our approach, we make use of a small program fragment depicted in Fig. 1. This fragment computes the minimum and maximum of a collection of integers stored in an array as well as the sum of the minimum and maximum under the pre-condition that the array comprises at least one element. Otherwise, an *Out of Bounds* exception would be raised when accessing the first element of the array in Line 2. Note, that changing this program in order to avoid the exception is simple, but increases the program size, which is less appropriate for explanation purposes. Moreover, because of the same reason and our assumptions, we exclude all definitions and type informations from the source code.

The program fragment comprises a bug in Line 13. In order to detect the faulty behavior we introduce a test suite comprising five different test cases (see Table 1). Each of them specifies values for the input variables and the expected output. When running our example program on each test case, the fragment returns unexpected values. So how to obtain the root cause for this detected misbehavior? Let us explain model-based debugging for extracting the root cause. For this purpose consider test case `A` from Table 1. When running the program on test case `A` only the following statements are executed:

```
 1.        i = 1;
 2.        min = input[0];
 3.        max = input[0];
 4.        while (i < length) {
12.        }
13.        result = min * max;
```

Let us now assume that each of these statements is represented by a relation (or mathematical equation) $R(v_1, \ldots, v_k)$ over the used variables $v_1, \ldots, v_k$ in that statement. Moreover, let us assume that each relation has an unique corresponding predicate $\neg AB_R$. A relation is used in a derivation if its corresponding variable is true. Formally, we represent this using the horn clause $\neg AB_R \rightarrow R(v_1, \ldots, v_k)$. For example, we represent statement `1.  i = 1;` using rule $\neg AB_1 \rightarrow i = 1$, where $i = 1$ is a relation stating that i has to be 1. We obtain similar rules for the other assignment statements. For simplicity and because of the fact that the while-statement is not executed, we ignore it. We describe the handling of such statements later in this paper.

The idea of model-based debugging is to use the set of obtained rules for debugging. We automatically obtain this set, which is the model, from the source code of the program. Hence, there is no need to manually construct the model. An explanation, i.e., a root cause, for a test case, which is called a diagnosis, is an assignment of truth values to the $\neg AB_R$ predicates such that the model together with the test case is satisfiable. Note that we represent the test case itself as set of relations.