Extending Diagnosis to Debug Programs with Exceptions

Wolfgang Mayer Markus Stumptner
University of South Australia
Advanced Computing Research Centre
5095 Mawson Lakes SA, Adelaide, Australia
{mayer,mst}@cs.unisa.edu.au

Abstract

Even with modern software development methodologies, the actual debugging of source code, i.e., location and identification of errors in the program when errant behavior is encountered during testing, remains a crucial part of software development. To apply model-based diagnosis techniques which have long been state of the art in hardware diagnosis, for automatic debugging a model of a given program must be automatically created from the source code. This work describes a model that reflects the sequential execution semantics of the Java language, including exceptions and unstructured control flow, thereby providing unprecedented scope in the application of model-based diagnosis to programs. Notably, this approach omits the strict view of a component representing one statement of earlier work and provides a more flexible mapping from code to model.

1. Introduction

Debugging, i.e., detecting a faulty behavior within a program, locating the cause of the fault, and fixing the fault by means of changing the program, continues to be a crucial and challenging task in software development. Many papers have been published so far in the domain of finding faults in software, e.g., testing or formal verification [3], and locating them, e.g., program slicing [17] and automatic program debugging [9] work that is being continued in a series of workshops on debugging [6]. In the 1990s model-based diagnosis techniques [16] started to be examined for locating faults in software, originally for logic programs [4], later for the debugging of large-scale concurrent hardware designs written in VHDL [8], and more recently imperative programs [13].

This paper extends prior research on model-based diagnosis for locating bugs in programs written in mainstream programming languages (e.g. Java). The idea behind the model-based debugging approach is (1) to automatically compile a program to its logical model or to a constraint satisfaction problem, (2) to use the model together with test cases and a model-based diagnostic engine for computing the diagnosis candidates, and (3) to map the candidates back to their corresponding locations within the original program. Formally, given a set of test cases on which the program is run, a (minimal) diagnosis is defined as a (minimal) set of incorrectness assumptions $AB(C)$ on a subset $C \in \Delta$ of components $COMP$ in the program (usually statements) such that $\{AB(C)|C \in \Delta\} \cup \{\neg AB(C)|C \in COMP \setminus \Delta\} \cup SD$ is consistent [16]. Here, SD is a logical theory describing the program's behavior under the assumption that components work correctly, and $AB(C)$ expresses that the program part modeled by C is possibly faulty (*ABnormal*) and can show arbitrary effects. Since the computation depends on observations in terms of test case output, unlike formal verification approaches, no separate formal specification is necessary – everything but the test cases is computed automatically from the source code. Conversely, where verification model checkers produce counterexamples, the outcome of the diagnosis process are code locations. Model-based debugging thus complements, rather than replaces verification techniques.

The models presented in [10, 15] have successfully been applied in the Jade project to debug Java programs, with tests performed on small to medium sized Java programs together with their faulty variants and given test cases. A comparison of the models and their effectiveness relative to each other as well as compared to a normal interactive debugger was given in [13]. Whereas these methods have been applied to large scale programs in concurrent languages [8], so far, one important factor that has held up experimentation with "off the shelf" example programs in the Jade project

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

has been the omission of certain frequently used language constructs from the models, most notably exception handling. The hierarchical structure of the models renders them unsuitable for diagnosing programs which do not conform to the assumption of a "structured" control flow based purely on loop and if statements. This excludes a wide array of language features which includes structured exception handling constructs, recursive method calls, return statements, and jump statements. This paper addresses that issue and presents a different modeling approach, where the restrictions on program structure are relaxed in order to overcome the limitations of previous models. As a result it represents a significant step towards a model of a language that can cover arbitrary Java programs.

The paper is organized as follows: In Section 2 we describe the construction of the extended diagnostic model. Section 3 described the model dynamics, focusing on loops and method calls. Finally, we discuss the capabilities of the model.

2. Model Structure

The modeling algorithm is based on building a modular, multi-directional version of the classical control flow graph (CFG) [7], which represents an approximation of all possible execution paths. The nodes of this CFG represent *basic blocks* (i.e. a sequence of operations that always executes sequentially without branching), and the arcs represent the control flow between these blocks. The CFG is enhanced by data flow information and the basic blocks in each method are then transformed into a component network, where the components model the behavior of the corresponding program statements. The representation of the program execution as a constraint system provides means for computing values in both directions: forward and backward. This is essential for applying model-based debugging approaches, which rely on an effective theorem prover generating small conflict sets.

The automatic transformation from the Java source code to the model used for diagnosis proceeds in three steps. For space reasons we only give a brief overview, full references to the algorithms and literature used are given in [11].

Build CFG. The program is transformed into its CFG representation. Each method is represented as a directed graph with basic blocks, entry and exit points as nodes and all possible transitions between nodes as arcs.

Compute Data-flow Information. In this step the CFG is enhanced with additional data flow information, resulting in what is known as the *Static Single Information Form (SSI form)* [1]. Essentially, a program is in SSI form if every variable in the program is assigned exactly once and every use is reached by exactly one assignment. This is guaranteed by renaming the variables in the original program and introducing data flow ϕ functions at the beginning of each basic block that is reached via multiple paths in the control flow graph. To enable backward analysis, the SSI form adds complementary σ functions.

Heap data structures require special treatment. We follow [13] and partition the heap into disjoint structures, each representing the values of a particular instance variable of a class. This allows to build a model for programs accessing heap data structures, even if the objects that are accessed through references are not known statically.

Transform CFG into Component Representation. For each primitive element in the model (e.g. variable access, assignments, operators, method calls, etc.) a constraint is generated that simulates the effects of that element in both forward and backward direction. Constraints are grouped into diagnosis components, which allows for hierarchic diagnostic reasoning using different levels of granularity.

For every assignment to a variable a new model variable is generated and is subsequently used up to the point in the code where the variable is reassigned. A special placeholder component is inserted for a method call and loop statement that summarizes the effects of the method or loop (see Section 3).

The modeling of the control flow proceeds as follows. For each control dependence region a constraint variable is created that represents the control flow for the region. The domain of control variables is defined as $\{\top, \bot, ?\}$, with $\top$ meaning the region is definitely executed for the test case, $\bot$ representing the fact that a region is never executed, and for ? it is unknown whether the region is executed. Control variables are part of the precondition of the behavioral description of every component representing a statement that is contained in a block and can therefore be used to remove paths which are conflicting with a test case from the model simply by setting the associated control variable to $\bot$. The values of the control variables are determined by (1) statements which represent branches in the CFG, and (2) liveness components. To allow for case (1), the control variables for every successor region of a branch statement are set by the behavioral description of branch statements. For (2), liveness components ensure required properties of valid execution traces through the control flow graph are maintained during model execution. In particular, it must be satisfied that a region can be entered iff an exit transition

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

```
1  class Fac {
2    static int f(int x) throws OutOfRangeException {
3      if (x < 0) throw new OutOfRangeException();
4      else if (x == 0) return 1;
5      else return x * f( x - 1 );
6    }
7    static boolean exception = false;
8    static int main() {
9      int fac = 0;
10     try {
11       fac = f(3);
12     } catch(OutOfRangeException ex) {
13       exception = true;
14     }
15     return fac; } }
```

Figure 1. Example Program

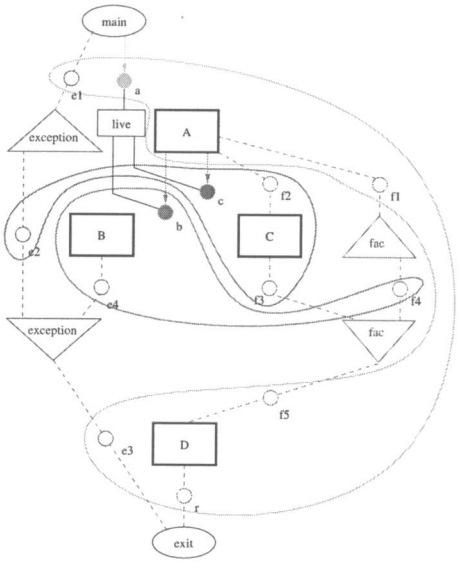


Figure 2. Model of Method main(). For clarity, blocks are drawn without embedded components. Model variables are drawn as circles, control variables are drawn filled. σ and ϕ components are represented by triangles. The set of blocks and variables associated with a control variable are drawn as hyperedges containing the control variable.

from the region is possible.

A formal model for basic block behavior is given in terms of logic formulas in [13]. Variables used in a statement s are added to the input connections of the constraint describing s, and variables assigned by the statement are added as output connections. A connection is used as input for subsequent statements until a new output connection for the corresponding variable is created.

Data flow between basic blocks is modeled by constraints representing ϕ and σ functions. ϕ functions combine the values $data_i$ of the incoming paths i into a new value for the output connection $data_u$ of the component. For σ functions, the behavior is reversed to duplicate an incoming connection for every successor region.

To combine the reasoning about data and control flow, each of the data flow variables is associated with the control region they are contained in. In case conflicting values are derived for a variable, the associated control region is marked unreachable and all contained constraints are disabled. A model is found inconsistent if all possible paths between the program's entry and exit statement are infeasible.

Example. To illustrate the modeling process, reconsider the program from Fig. 1. The model of method main is depicted in Fig. 2.

Control flow at the entry point of the method is represented by control variable a. The call of method f() splits the control flow in two separate paths. Therefore, σ components are inserted for the variables fac and exception. The output variables for the components representing fac is associated with the control variable of Block B. For exception, the variable is associated with Block C's control variable.

The path represented by variable b represents the transition to the exception handler block. The normal return from the method is represented by c. As exactly one of these paths has to be followed if Block A can be reached, the liveness constraint live enforces this condition. The model of Block D requires variable fac and the control flow as input connections. As there are multiple incoming arcs for both dependencies in the CFG, ϕ components have to be inserted for fac and exception (as they are modified in the subregions B and C). The components for the variables are connected to the two preceding components providing the values of the variable. Further, the output of the ϕ component is connected to Block D, which requires the value of fac.

3. Model Dynamics

To use the model for diagnosis, the behavior of all model constraints has to be specified in a form suitable for diagnosis. For brevity, we focus on the most

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

interesting constructs, namely loops and polymorphic method calls, and refer to [11, 10] for the discussion of other components.

Loops are modeled as hierarchical components, where the loop component contains separate subcomponents for the loop's condition and the body. A copy of the loop's condition is inserted into the model before the loops component, to determine if the loop should be executed. If the condition evaluates to false, the loop component is removed from the model and its output connections are replaced with its input connections.

In case the condition evaluates to true, the component is expanded and copies of the models of the body and the condition are inserted before the component (in this order). Inlining the loop's components into the control region allows for fine-grained diagnosis of the loops statements, possibly resulting in significantly improved diagnostic accuracy compared to previous models.

It also has to be taken into account that the loop might not terminate for particular input configurations. As our systems assumes termination of the program, this constitutes a conflict and the branch leading to this configuration is removed by marking the region containing the loop as unreachable. To detect nontermination, the system currently assumes an (user-specified) upper bound on the number of iterations each loop may execute. In the future, this could be enhanced with more advanced analysis, such as Abstract Interpretation [5].

In case the loop condition cannot be evaluated to a unique result, we approximate the behavior of the loop by bidirectional Abstract Interpretation [2]. This approach allows to exploit the values derived by program (reverse) execution for the input (and output) values of the loop to obtain enhanced accuracy compared to completely static analysis. Also, the dynamic model modification allows for determining the amount of context that is considered in the analysis to vary depending on the actual values computed during model execution. For example, consider the following program (the environment before loop execution is denoted in angled brackets):

$\langle a \leftarrow [-5, -3, 2, 0, 1], i \leftarrow 0, s \leftarrow 0 \rangle$
while(...) {
  s = s + a[i]
  i = i + 1
}

Using simple interval abstraction [5], and the fact that the array access expression constrains the values of i, together with the monotonicity of i, we derive that the number of loop iterations is bounded by 5. Expanding the loop into five nested selection statements, the value of s after the loop is approximated by $s \in [-25..5]$, which may be used to derive subsequent inconsistencies.

Method calls are modeled similarly to loop statements, with multiple alternate subcomponents for polymorphic calls, each representing a possibly called method. When the components is executed, the appropriate model is selected using the actual type information of the receiver. Models of other methods are removed.

4. Model Properties

We close with an overview over the impact of some modeling decisions on the diagnostic process.

Conflicts. The model derives an inconsistency if and only if there is no possible and consistent path between the model's start node and its end node. In this case, to derive conflicts, the propagation algorithm must keep track of which components derive which values.

For efficiency, the component system propagates only deltas of values, so that execution in the next propagation cycle can be focused on the newly added values. Also, method calls and loops are expanded to full models only when reasonable input values are present at their inputs (e.g. only if the receiver is known).

Faults. The model proposed herein is able to locate general functional faults, where parts of a program are faulty without assigning to wrong variables. This also includes the subclass of structural faults where a wrong variable is *read*. Faults can be detected not only in the top level method, but also in hierarchically embedded program elements. However, the model assumes the control flow graph to be fixed and therefore faults causing changes in the CFG cannot be located reliably. For example, changing the type of the exception of a throw statement may cause a changed control flow, as a different catch block may be responsible for the new type.

5. Conclusion

Model-based debugging carries considerable promise based on the concept of automatically deriving a diagnosis model from the program source code and using this, together with input-output test vectors, to identify faults in programs [14]. An effective model representation is a crucial step in building a framework for model-based debugging. Our work represents

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

the first time that non-structured control flow is represented in a model-based diagnosis scheme. We have described the computation of the bidirectional model that permits reasoning about unstructured control flows in both directions as required for diagnostic reasoning. Further, the dynamic expansion of the model to accommodate loops and method calls has been described. Notably, this approach omits the strict view of a component representing one statement of earlier work [13] and provides a more flexible mapping from code to model.

Since real-world programs make frequent use of constructs such as exceptions, this model represents a significant step beyond earlier models that represented Java programs for debugging, but were restricted to linear control flows. As it shares the basic model-based architectures, it can be smoothly incorporated into the implemented JADE intelligent debugging engine [13].

Ongoing and future research includes extending the model to handle abstract program properties and incorporate additional design information to locate structural faults and missing statements [12]. Also, we are currently optimizing our algorithms to construct the model dynamically and include only the actually executed parts of the program. Finally, evaluating our prototype using a set of realistic test cases and developing effective heuristic strategies to focus on the most promising fault candidate are ongoing and open research issues.

References

[1] S. Ananian. The static single information form. Master's thesis, Department of Electrical and Computer Science, Princeton University, 1999.

[2] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 46–55, 1993.

[3] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, Sept. 1994.

[4] L. Console, G. Friedrich, and D. T. Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*, pages 1494–1499, Chambery, Aug. 1993.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, Jan. 1977.

[6] M. Ducasse, editor. *Proceedings of the 4th International Workshop on Automated and Algorithmic Debugging, AADEBUG '00*. Munich, 2000.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[8] G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.

[9] J. W. Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5:133–154, 1987.

[10] C. Mateis, M. Stumptner, and F. Wotawa. A Value-Based Diagnosis Model for Java Programs. In *Proc. 11th Int'l Workshop on Principles of Diagnosis*, Morelia, Mexico, June 2000.

[11] W. Mayer and M. Stumptner. Debugging program exceptions. In *Proceedings of the Fourteenth International Workshop on Principles of Diagnosis*, Washington, D.C., June 2003.

[12] W. Mayer and M. Stumptner. Model-based debugging using multiple abstract models. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging, AADEBUG '03*, Ghent, Sept. 2003. Accepted for publication.

[13] W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa. Can AI help to improve debugging substantially? Debugging Experiences with Value-Based Models. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 417–421, Lyon, 2002.

[14] W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa. Towards an Integrated Debugging Environment. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 422–426, Lyon, 2002.

[15] W. Mayer, M. Stumptner, and F. Wotawa. Model-based Debugging or How to Diagnose Programs Automatically. In *Proc. IEA/AIE*, Springer LNAI, pages 746–757, Cairns, Australia, June 2002.

[16] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[17] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY