# High-Level Observations in Java Debugging

**Wolfgang Mayer** and **Markus Stumptner**[1]

**Abstract.** Recent years have seen considerable developments in modeling techniques for automatic fault location in programs. However, much of this research considered the models from a standalone perspective. Instead, this paper focuses on the properties of the testing and measurement process, where capabilities differ strongly from the classical hardware diagnosis paradigm. In particular, in an interactive debugging process user interaction may result in highly complex input to improve the process. This work proposes an heuristic entropy-based measurement selection algorithm, which incorporates high-level properties of the intended behavior of Java programs, specific to a set of test cases. We show how to integrate the approach into previously developed model-based debugging frameworks and to how reasoning about high-level properties of programs can improve fault localization.

**INTRODUCTION** 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 $TC$ 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 $\{AB(C)|C \in \Delta\} \cup \{\neg AB(C)|C \in COMP \setminus \Delta\} \cup SD \cup TC$ is consistent [7]. Here, $SD$ is a logical theory describing the program's behavior under the assumption that components work correctly, $TC$ are the values specified by a test case, and $AB(C)$ expresses that the program part modeled by $C$ is possibly faulty (*AB-normal*) 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. Our more recent work (e.g., [5]) has added models based on the Abstract Interpretation Framework [1] and also moved to more efficient models that are based on test case specific representation of individual traces.

**DEBUGGING WITH HIGH LEVEL OBSERVATIONS** While considerable improvements in the modeling and diagnostic algorithms have been achieved, the interactive aspect of (semi-) automatic debugging has so far taken second seat behind the computational aspects. In particular, previous research prototypes combined a standard debugger-like interface with a variant of the standard entropy-based selection of measurements to identify points during program execution where the debugger user would be queried about the correctness of (parts of) the program status at that point in execution (the user serving as "oracle"). The experience was that answering these oracle queries posed by the system could be difficult in many cases. To be useful for interactive debugging, an approach to queries is needed that is both more powerful and simple for the developer (debugger user) to apply. We introduce the notion of *high-level observations* (HLO's) about the expected behavior of the program. Dedicated HLO predicates provide high level descriptions of program execution beyond the classical diagnosis test of whether a given part of the program state is correct or incorrect. We refer to an observation as high-level if it constrains multiple program states and/or locations. HLO's thus allow for debugging capabilities beyond current modeling approaches while keeping the information that has to be provided by the user at a minimum.

HLO's are produced by presenting queries to the user that have been ranked high as measurements to be selected (see below); the query specifies a high level condition and the user provides a HLO, or measurement outcome, by answering it. A query $Q$ is a condition on the program state or actual execution that divides test cases $T$ into sets $R_T$ and $R_F$ that predict $Q$ to be true and false, respectively. Candidates in $R_U$ are those that do not predict a value for $Q$.

When debugging using a model that follows the program semantics closely, three main causes of complex queries can be identified. First, unless the observed variable is at a point close to the start or the end of the program execution, the user needs to simulate much of the programs behavior to compute the desired value. Second, frequent switching between different execution states makes it much harder for the user to build a model of the correct execution of a program. This is especially true if the execution states are deep in the middle of some complex computation. Finally, the user cannot rely on values of variables provided by the debugger, as these may have been influenced by the true program fault or the diagnostic assumptions. Using HLO's, the ability to obtain a description of *intent* from the user (developer) makes observations potentially more powerful than with hardware. Types of HLO's include:

**Traversal properties.** Elements of arrays and dynamic data structures are often processed such that either all of them are read or updated. If the underlying data structure is monotonic, this allows to bound the number of times the iterating loop or recursive function is executed.

**Read- and write-only.** Read-only assertions allow to ignore any attempted update to data structures passed to a loop or method call, and exclude diagnostic candidates implying such updates. Write-only access is used to decouple the previous from updated values of a data structure.

**Subproblem (in)dependence.** Loops and recursive method invocations can be modeled differently if it is known that computations

[1] University of South Australia, Advanced Computing Research Centre, Mawson Lakes, SA 5095, Adelaide, Australia. E-mail: {mayer,mst}@cs.unisa.edu.au.

in different loop iterations and disjunct sub-structures of dynamic data structures are independent.

**Variable (in)dependence** can be utilized to infer missing statements, uses and updates of wrong variables, or shortcut conflict computation.

**Loop specific invariants.** Loops based on counters or other induction variables [4] can often be bounded if monotonicity of the induction variable is assumed. Although it is possible to infer that property for large classes of loops using syntactic pattern-based and Abstract Interpretation [1] approaches, manual specification also eliminates the fault candidate where the update expression of the induction variable is assumed abnormal.

**UML invariants.** Invariants taken from UML class diagrams, such as type constraints and cardinality constraints for relations.

**Region reachability.** Statements that should be executed always (or never, or at least once) are marked to remove paths that would otherwise contribute to spurious fault candidates.

**Generating High Level Queries** Queries about high-level properties are generated using an approach borrowed from the Daikon [3] "potential invariant" detection tool. A set of properties (user-defined and built-in) are tracked while test cases are executing, and all "invariants" eliminated that do not imply all properties. Statistical measures are used to discard invariants without sufficient support.

Daikon only supports forward execution, but our approach also allows backward reasoning and variables with unknown values to support invariant detection even if fault assumptions are present. HLO's are instantiated into queries if there is a test case $T$ and a fault candidate that predicts $\mathsf{false}$ for the tracked property $P$ given $T$.

**UTILIZING INSTANTIATED INVARIANTS** The prevailing use for *inferred invariant observations* (IIO's) is to generate additional conflicts, which eliminate diagnosis candidates. Therefore, we infer IIO's from the correct test cases and instantiate queries only if a particular fault candidate $D$ violates the invariant for a test case. This is restricted further such that all queries are discarded where the invariants of all fault candidates agree for a test case. The aim is to avoid uninformative queries in case the invariant is too strong due to an insufficient number of correct test cases. IIO's are also represented as queries.

**SELECTING MEASUREMENTS** A solution for selecting good measurements given a test case and a set of fault candidates was presented in [2]. The algorithm utilizes entropy to find the variable that, when observed, on average eliminates the most candidates. Only fault probabilities for components and the values predicted by the fault candidates are required. For the software domain, we obtain fault probabilities from the execution paths of correct and faulty test cases [6]. Components that are executed for few correct test cases and many failing test runs are assigned a higher fault probability.

To integrate the HLO's and the IIO's into the measurement selection, we define auxiliary variables $o_i$ for each query $Q_i$, with $\mathsf{dom}(o_i) = \{true, false\}$. The sets $R_T$, $R_F$, and $R_U$ are used to compute the entropy for $o_i$: $H(o_i) = p(o_i = true) \log p(o_i = true) + p(o_i = false) \log p(o_i = false) + p(R_U) \log 2$ ($H$ denotes the entropy, $p(o_i = v) = p(S_v) + \frac{p(R_U)}{2}$, where $S_v$ is the set of selected candidates where $o_i = v$, and $p(X)$ denotes the summed probability of all fault candidates in $X$). Finally, the measurement with maximal entropy is selected.

This approach (on average) optimally discriminates fault candidates but proves insufficient for interactive debugging, mainly due to the queries being too complex to answer with reasonable effort. For interactive debugging, selected queries must also conform to the following properties: (1) Queries that are deep in the execution trace of a program are difficult to answer. (2) Selected queries should not "jump around" in the execution trace, as this prohibits the user from building a model of the correct execution of the program. (3) Subsequent queries should use the same test case, if possible.

To elude these problems, we propose to extend the entropy-based measurement selection with a heuristic approach.

To minimize query complexity, for each query we compute a "distance" $d_i$ between the location tested by the query and the location of the closest answered query (or the program start or end point) for the same test case. $d_i$ is derived from the execution profile of the test cases and all the fault candidates: Each statement that is executed for at least one fault candidate is marked. Also, the union of all the call graphs for each fault candidate is computed.

Starting at $l_i$, transitions between statements are explored to find the path to an answered query $Q_j$ where the difference between the minimal nesting depth and the maximal nesting depth is minimal. To find the closest query, we apply a simple best-first search algorithm, following only transitions between marked statements. The nesting depth of a statement in a method is computed from the source code. For called methods, all possible call and return transitions are followed, according to the call graph generated earlier. The nesting depths for each method are summed.

**FURTHER WORK** The approach described in this paper has provided promising results on a set of small "toy" programs, but considerable issues remain for further work. Currently, the measurement selection is not fully integrated in our debugging prototype, which makes further evaluation difficult. Further, it is not clear whether the parameters for the heuristic query search must be preset or if there exist good heuristics to choose and update those values. The user interaction aspect of interactive debugging also requires more investigation, in particular, the question on what and how much context needs to be provided to the user to allow efficient query answering. In contrast to other debugging and verification approaches our method has the advantage that it does not require to specify the behavior of the program in a formal language. Rather, it is sufficient to provide properties and invariants that are *specific to a set of test cases*, which is usually much easier, especially as the behavioral description need not be complete. Also, complexity is lower than for verification, as we do not follow *all* possible executions of a program. Instead we focus on the program behavior specific to a set of test cases, which is usually good enough if the test set is large.

## REFERENCES

[1] Patrick Cousot and Radhia Cousot, 'Abstract interpretation based program testing', in *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, (2000).

[2] Johan de Kleer and Brian C. Williams, 'Diagnosing multiple faults', *Artificial Intelligence*, **32**(1), 97–130, (1987).

[3] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin, 'Quickly detecting relevant program invariants', in *Proc.ICSE-00*, pp. 449–458, Limerick, Ireland, (June 7–9, 2000).

[4] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe, 'Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA', *ACM TOPLAS*, **1**(17), 85–122, (1995).

[5] Wolfgang Mayer and Markus Stumptner, 'Model-based debugging using multiple abstract models', in *Proceedings of the* $5^{th}$ *International Workshop on Automated and Algorithmic Debugging, AADEBUG '03*, pp. 55–70, Ghent, (September 2003).

[6] Wolfgang Mayer, Markus Stumptner, Dominik Wieland, and Franz Wotawa, 'Towards an Integrated Debugging Environment', in *Proc. ECAI*, pp. 422–426, Lyon, (2002).

[7] Raymond Reiter, 'A theory of diagnosis from first principles', *Artificial Intelligence*, **32**(1), 57–95, (1987).