Debugging Program Loops using Approximate Modeling

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

Abstract. Developing model-based automatic debugging strategies has been an active research area for several years. We analyze shortcomings of previous modeling approaches when dealing with object-oriented languages and present a revised modeling approach. We employ Abstract Interpretation, a technique borrowed from program analysis, to improve the debugging of programs including loops, recursive procedures, and heap data structures. Together with heuristic model refinement, our approach delivers superior results than the previous models. The principle of our approach is demonstrated on a running example.

1 INTRODUCTION

Developing tools to support the software engineer in locating bugs in programs has been an active research area during the last decades, as increasingly complex programs require more and more effort to understand and maintain them. Several different approaches have been developed, using syntactic and semantic properties of programs and languages. A common drawback of these tools is the high user interaction required to locate faults. For that reason, model-based debugging (MBD) was introduced in [4], where the authors show that model-based techniques are capable of providing results using less interaction than previous approaches, and approach that was later extended to imperative and finally to object-oriented languages.

This paper extends past research on MBD of mainstream object oriented languages, with Java as concrete example [9]. We show how abstract program analysis techniques can be used to improve accuracy of results to a level well beyond the capabilities of past modeling approaches. In particular, we discuss adaptive model refinement and automatic selection of the 'most promising' fault candidates. Our model refinement process targets loops in particular, as those were identified as the main culprits for imprecise diagnostic results. We exploit the information obtained from abstract program analysis to generate a refined model, which allows for more detailed reasoning and conflict detection. Finally, heuristics comparing diagnoses and their corresponding model traces are developed to effectively select promising diagnoses. Subsequently, those can be used to selectively generate a refined model to narrow down possible fault locations or detect different kinds of faults, for example missing statements.

While the proposed approach overcomes many problems of previous approaches, using a fixed abstraction may be unsuitable to prove inconsistency for some programs. Custom abstractions as in model checking (although computationally expensive) are future topic. As our analysis is bidirectional, a trace of the program execution must be kept which may be infeasible for large programs. Programs interfacing with external systems, such as file systems, cannot be handled without providing suitable abstract models for these interfaces. Finally, while we can debug programs with structural faults (e.g., using wrong variables) at method level, locating such faults at statement level requires further information that must be provided by the user through additional observations and complementary models [13].

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

Section 2 introduces to the basics of MBD. Section 3 summarizes abstract interpretation and introduces our conflict detection mechanism. Section 5, introduces a model refinement. Finally, we discuss relevant related work and the current status of our prototype.

2 Model-based Debugging

To locate faults using model-based diagnosis techniques, the source code of the program P to be analyzed must be available. Also, a set of test cases $\mathcal{TC}$ is required, which (partially) specify the expected behavior of P. The test case descriptions are not limited to concrete values. Our approach also allows to utilize more abstract observations; for example, assertions, reachability constraints for some paths, valid call sequences, acyclicity of data structures, aliasing properties, etc.

The connection from P's source to the model is realized through a set $COMPS$ and a set $\mathcal{M}$ of models. $COMPS$ contains the set of components of which fault candidates are composed, each representing a particular part of P. As an example, to locate individual faulty statements, each statement $s_i \in P$ is associated with a component $c_i \in COMPS$.

Each model $m \in \mathcal{M}$ describes the program behavior (possibly at an abstract level) and is automatically generated from the source. For each element of the programming language, there exists a model fragment that describes the element's behavior, as is specified in the language specification. This description is multi-directional, which allows us to reason backward through P to detect conflicts. Also, each component has an "abnormal" mode, denoted $ab(\cdot)$, which denotes possible faults (as opposed to the "correct" behavior, denoted $\neg ab(\cdot)$). The model of P is obtained by composing instances of model fragments according to the program structure.

To find possible faulty components we employ the consistency-based approach [12]. As each component corresponds to a particular part of the program, every diagnosis can be mapped back into P to indicate possible faults to the programmer.

Example 1 *To illustrate the approach, the following Java fragment is modeled using the value-based approach presented in [9]:*

```
1    int x = 2 * a;
2    int y = x * 3;
3    int z = 5 * x;
```

The model represents the program as a constraint network. For each statement, there is a component modeling the effect of that statement. The components are composed of simpler constraints, each of which represents a sub-statement, which cannot be part of a diagnosis on its own. The components and constraints are connected according to the data flow. A graphical representation of the model of the above program is presented in Figure 1. Together with a test case that specifies $a \leftarrow 1, x \leftarrow 2, y \leftarrow 3, z \leftarrow 10$, $ab(\boxed{y=x*3}_2)$[2] is identified as the only single fault. Mapped back to the program, this corresponds to a fault in line 2.

While dependency- and value-based program models have been successful in locating faults in imperative programs, these models are created without the aid of test case information and thus have to

[2] Components are denoted by the source code fragment that they represent, together with the line number.
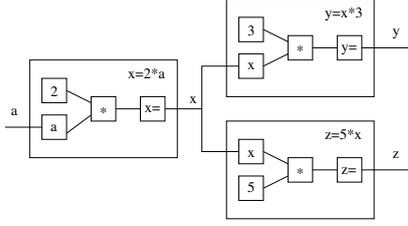
Figure 1. Example Model

account for all possible cases. In object-oriented languages, this can lead to large models:

Example 2 1 Object o = list.iterator().next();
2 String s = o.toString();

*If **list** is an instance of some class implementing the Java **List** interface, all possible implementors of method **iterator()** have to be modeled. The same holds for method **next()**, as the return type of **iterator()** is also an interface. What's more, the call to **toString()** has to account for all possible implementations of the method, as **Object** is a superclass of all Java classes. Also, in Java, each dereference of a **null** reference at runtime causes an exception, which accumulates to a very dense and awkward to handle control flow graph.*

Similar experiences with static analysis of heap data structures [11] have shown that even if only a few objects are actually used during a program run, the statically determined information can be very large and imprecise. To overcome these problems, we propose to intertwine model construction and conflict detection to obtain concise and test-case specific models, which allow for more efficient reasoning and conflict detection.

Another problem of the test case independent models is the handling of loops in case the number of loop iterations cannot be determined.[3] Should this case arise, earlier models [9] could not propagate any information across the loop, resulting in a large number of diagnoses. To overcome that deficiency, we combine static program analysis techniques, program execution, and heuristic abstraction to infer approximations of values before and after the loop.

The idea is to use Abstract Interpretation [6] to generate a trace of the program when executed with inputs from a test case. While the trace is constructed, only the paths that are found feasible using an abstracted version of the concrete program state are constructed; other paths are ignored, leading to concise model representations. As the trace is constructed dynamically, it is easy to insert and check any values, invariants, assertions, or other constraints provided by the test case. A model is found inconsistent if there is no feasible program path between the program's initial state and the point where it completes normally. A conflict is extracted from the inconsistent trace by collecting all components that are required to obtain the inconsistency. Finally, a standard model-based diagnosis algorithm is applied to find possible diagnoses given the conflict(s).

3 ABSTRACT INTERPRETATION

The basic idea of Abstract Interpretation [6, 2] is to replace the concrete semantics $S[\![P]\!]$ of a program P, which expresses the exact effects of P (which are not computable in general) with an abstraction $S^{\#}[\![P]\!]$ to obtain a finite representation which can be computed automatically. $S[\![P]\!]$ and $S^{\#}[\![P]\!]$ each operate on a domain representation, D and $D^{\#}$, respectively. D and $D^{\#}$ are usually represented as lattices $(\mathcal{P}(S), \emptyset, S, \subseteq, \cup, \cap)$ (S denotes the set of program states) and $(\mathcal{P}^{\#}(S), \bot, \top, \sqsubseteq, \sqcup, \sqcap)$, respectively, where the latter is a computable abstraction of the former. $\bot$ denotes infeasibility and $\top$ represents all possible states. $\sqsubseteq$ represents the abstract state ordering,

and $\sqcup$ and $\sqcap$ denote the least upper bound and greatest lower bound operator, respectively. The key component in this framework is a pair of functions (α, γ) ("Galois connection"), where α maps sets of concrete program states to their representation in the abstract domain $D^{\#}$ and γ maps each abstract state to its meaning in D.

$S^{\#}[\![P]\!]$ can then be expressed as fixpoint over a set $\mathcal{X}$ of equations derived from P's source code. The equations are composed of abstract operations $\Phi^{\#} \equiv \gamma \circ \Phi \circ \alpha$, which model the effects of every operation Φ in P. An approximation of the forward semantics $S_{+}^{\#}[\![P]\!]$ is given by the solution of lfp $\lambda X \cdot (E \sqcap \mathcal{X}(X))$ (starting at $\bot$), where E denotes the approximation of the entry states (in our case provided by the test case).

In contrast to conventional static analysis, where the equation system X models all possible paths in P, we construct the system for a specific test case (and mode assumptions). This allows us to eliminate all paths that are infeasible for the specific test case. The reduced number of paths does not only contribute to more concise models, it also helps to reduce the number of expensive join operations that are necessary whenever a program point is reachable via multiple paths.

In case the abstract lattice is of infinite height, narrowing (Δ) and widening (∇) operators have to be applied to ensure termination of the computation [6]. Widening operators selectively discard information from the abstract states to guarantee that the computation of mutually dependent equations eventually converges in a finite number of iterations. Narrowing operators are used to eliminate ∞ in certain cases after a post-fixpoint has been found. Backward semantics are analyzed in a similar manner in [2], by insertion of *invariant assertions* ("always"), which ensure that every time the assertion is reached, the condition is satisfied, and *intermittent assertions* ("eventually") which guarantee the condition is true at least once. We exploit these assertions to eliminate unwanted paths from our traces. For example, the intermittent assertion **eventually true** at the program exit ensures that only terminating traces are considered. Similarly, invariant assertions can be used to eliminate all uncaught exceptions by providing an exception handler that must not be reached.

A sequence of forward and backward reasoning steps can be defined to approximate the entry and exit states which guarantee the validity of the assertions [2], leaving only those traces that satisfy all assertions. Thus if $\bot$ is derived for the program entry state, it is certain that no feasible execution exists from the program entry point, and a conflict between the assertions and the program is derived.

In this work, we use a non-relational variant of the interval abstraction to approximate a set of numbers. Formally, the Galois connection is defined as follows:

$\alpha(\{x_1, \ldots, x_n\}) = [\min(\{x_1, \ldots, x_n\}), \max(\{x_1, \ldots, x_n\})]$
$\gamma([a, b]) = \{x \mid a \leq x \leq b\}$

$$[a_1, b_1] \nabla [a_2, b_2] = \begin{bmatrix} \textbf{if } a_2 < a_1 \textbf{ then } -\infty \textbf{ else } a_1, \\ \textbf{if } b_2 > b_1 \textbf{ then } +\infty \textbf{ else } b_1 \end{bmatrix}$$

Sets of object references beyond user-specified size threshold are abstracted by selectively merging references according to the program structure (e.g., merge values of objects that are created by the same statement or the same procedure). In loops where the number of iterations cannot be determined, a common summary object is always used for all objects created by a particular statement.

Example 3 *Consider the program in Figure 2. Assume that the diagnosis engine believes $\{ab(\boxed{i=0}_9)\}$.[4] Then, the number of iterations for the loop in line 9 cannot be determined exactly. Without special treatment of loops (see Section 5), the environments before the loop is entered/exited include (among others) the following values after the fixpoint has been reached:*

---

[3] Note that although a test case may specify all the information necessary to execute the program, the diagnostic engine may introduce fault assumptions which invalidate some values and give rise to several possible program runs.

[4] For brevity, we omit components that behave normally.

```
1    class Item {
2      int value;
3      Item(int v) {
4        value = v;
5      } }
6    class Bag {
7      Item[] items = new Item[5];
8      void add(Item item) {
9        for (int i = 0; i < items.length; i++)
10         if (items[i] == null) {
11           items[i] = item;
12           return;
13         }
14       assert false; }
15     void removeCheaperThan(int value) {
16       int i = 0;
17       while (i < items.length) {
18         if (/* items[i] != null &&*/ items[i].value < value)
19           items[i] = null;
20         ++i;
21       } }
22     static void demo() {
23       Bag b = new Bag();
24       b.add(new Item(10));
25       b.add(new Item(20));
26       b.add(new Item(25));
27       b.removeCheaperThan(20);
28       assert b.items[0] == null
29         && b.items[1].value == 20
30         && b.items[2].value == 25
31         && b.items[3] == null
32         && b.items[4] == null;
33  } }
```

Figure 2. Example Program and Test Driver

	line 9	line 12
i	$\top$	$[0,4]$
this	$\{o_{23}\}$	$\{o_{23}\}$
o_{23}.*items*	$\{o_7\}$	$\{o_7\}$
o_7.*length*	$[5,5]$	$[5,5]$
$o_7.0 \cdots o_7.4$	*{null}*	$\{o_{24}, o_{25}, o_{26}, null\}$

The o_k denote arbitrary, unique object IDs generated during trace construction, k is the line number where the object was allocated.

4 MODEL CONSTRUCTION

Our Abstract Interpretation based debugging framework seeks to avoid building a complex model representing the whole program. Rather, the abstract interpreter is used as a consistency checker, given a program P and a test case T.

1. T is compiled into a test driver program P_T and all assertions specified in T are (temporarily) inserted into P. Note that no components are created for assertions, as these are assumed to be correct. To guarantee finite traces, we consider nonterminating loops as errors and enforce a user-specified maximum iteration limit.

2. Construct initial trace using the forward semantics $S_+^{\#}(P_T)$ starting with an empty environment (the input values are specified in T and are compiled into the test driver in form of assertions and assignments).

3. Analyze trace backward to enforce the invariant assertions, eliminating all values that would lead to paths that do not satisfy the assertion conditions.

4. Analyze backward to remove all values implying paths that do not satisfy the intermittent invariants

5. Analyze forward to eliminate paths that are no longer feasible.

6. Repeat from step 3 until a fixpoint has been reached.

In any step, if the infeasible environment $\bot$ is derived for either the program entry point or the program exit point, no feasible path exists. The algorithm stops and a contradiction between P and T has been detected. Otherwise, P and T are not found inconsistent.[5]

---
[5] Note that this does not imply that P and T are actually consistent, as the used abstraction may be too weak to prove that.

Example 4 *Consider the program in Figure 2 and the test driver method* demo(). *When run, the program crashes with an exception due to an omitted* null *check in line 18. Assume our debugger has created a component for each statement of the program, with the exception of the test driver, which is assumed to be correct.*

For brevity, we skip the first trace construction, as all statements are assumed normal and the trace is simply a sequence of environments. At line 18, the thrown NullPointerException *causes a contradiction and the resulting conflict is* $\{\neg ab(\boxed{i=0}_{16}),$ $\neg ab(\boxed{i<items.length}_{17}),$ $\neg ab(\boxed{items[i].value<value}_{18}),$ $\neg ab(\boxed{items[i]=null}_{19}), \neg ab(\boxed{++i}_{20})\} \cup \{$ *"all statements in method* add()"}. *(See below for details on conflict derivation.)*

Assume the diagnosis engine tries to remove the conflict by assuming $\{ab(\boxed{i=0}_{16})\}$ *first. Again, the trace construction proceeds trivially up to the point before line 16. Here, the ab mode for the assignment forces the value of* i *to* $\top$, *resulting in the environment:*

this	$\{o_{23}\}$	i	$\top$	$o_7.2$	$\{o_{26}\}$
o_{23}.*items*	$\{ob_7\}$	$o_7.0$	$\{o_{24}\}$	$o_7.3$	*{null}*
o_7.*length*	$[5,5]$	$o_7.1$	$\{o_{25}\}$	$o_7.4$	*{null}*

As the loop condition cannot be evaluated uniquely without the exact value of i, *the analyzer constructs a cyclic trace containing the loop condition and the loop body and iterates the computation until a fixpoint is reached. The final trace for the loop is depicted in Figure 3. For space reasons, only the variables* i *and the array* o_7 *(accessed through the variable* items) *are depicted. The first array element is possibly* null *in the loop, as the abstraction cannot determine the proper index for the assignment in line 19. (The* null *values for indexes* 2 *and* 3 *are excluded through backward reasoning from the assertions in the test driver.) Note that the condition in line 18 does not evaluate uniquely for the same reason. Therefore, it cannot be proved that an exception is always thrown and both paths remain in the trace, circumventing conflict detection. Consequently,* $\{ab(\boxed{i=0}_{16})\}$ *is a single fault diagnosis.*

An environment represents the abstract state of the program at a certain point of execution, expressed by mappings from variables to (abstract) values, for local and global variables and heap data structures. The latter is critical as the number of objects allocated in a loop cannot be determined if the number of iterations is not known precisely. In this case, a summary object is created, conservatively approximating all objects allocated by the same statement in the loop.

Definition 1 *A program trace is a tuple* $\langle E, R \rangle$, *where E denotes a set of environments and $R \subseteq \gamma(E) \times \gamma(E)$ the transfer relation that is derived from program P and specifies the possible transitions between (concrete) environments, i.e., paths in the trace and the effects of the executed statements on the environments.*

Given a trace $\langle E, R \rangle$, each element of R is annotated with a component identifier used to extract conflict sets from the trace. Conflicts are found by tracking the smallest set of components that are used to infer each value in an environment. The components are trivially obtained from the relation used to compute the value. Should there be more than one set of possible explanations for a value, we keep the first of the smallest sets. While this does not guarantee minimality of the conflicts, preliminary experiments showed that this strategy almost always leads to minimal conflicts.

The component annotations also form the connection between the trace construction and the diagnosis engine. Depending on the component mode, different relations are instantiated, which forces the trace to realize different component behaviors. The implementation of the transfer relations in R is abstraction specific and has to be provided by the implementation of the abstract interpreter or the user.

If the program includes components where different modes may modify different variables (e.g., assignments to fields may modify different objects when abnormal), an extended algorithm is applied:
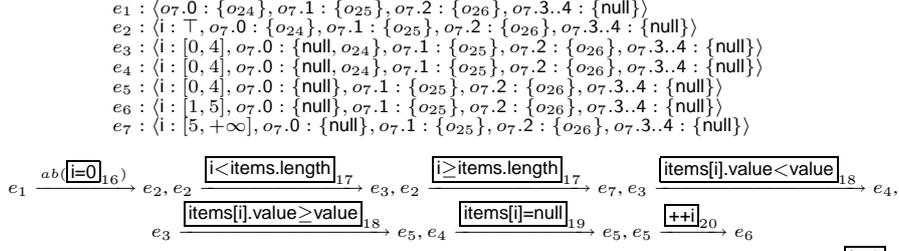
$$e_1 : \langle o_7.0 : \{o_{24}\}, o_7.1 : \{o_{25}\}, o_7.2 : \{o_{26}\}, o_7.3..4 : \{\text{null}\}\rangle$$
$$e_2 : \langle i : \top, o_7.0 : \{o_{24}\}, o_7.1 : \{o_{25}\}, o_7.2 : \{o_{26}\}, o_7.3..4 : \{\text{null}\}\rangle$$
$$e_3 : \langle i : [0,4], o_7.0 : \{\text{null}, o_{24}\}, o_7.1 : \{o_{25}\}, o_7.2 : \{o_{26}\}, o_7.3..4 : \{\text{null}\}\rangle$$
$$e_4 : \langle i : [0,4], o_7.0 : \{\text{null}, o_{24}\}, o_7.1 : \{o_{25}\}, o_7.2 : \{o_{26}\}, o_7.3..4 : \{\text{null}\}\rangle$$
$$e_5 : \langle i : [0,4], o_7.0 : \{\text{null}\}, o_7.1 : \{o_{25}\}, o_7.2 : \{o_{26}\}, o_7.3..4 : \{\text{null}\}\rangle$$
$$e_6 : \langle i : [1,5], o_7.0 : \{\text{null}\}, o_7.1 : \{o_{25}\}, o_7.2 : \{o_{26}\}, o_7.3..4 : \{\text{null}\}\rangle$$
$$e_7 : \langle i : [5,+\infty], o_7.0 : \{\text{null}\}, o_7.1 : \{o_{25}\}, o_7.2 : \{o_{26}\}, o_7.3..4 : \{\text{null}\}\rangle$$

$$e_1 \xrightarrow{ab(\boxed{i=0}_{16})} e_2, \quad e_2 \xrightarrow{\boxed{i<items.length}_{17}} e_3, \quad e_2 \xrightarrow{\boxed{i\geq items.length}_{17}} e_7, \quad e_3 \xrightarrow{\boxed{items[i].value<value}_{18}} e_4,$$
$$e_3 \xrightarrow{\boxed{items[i].value\geq value}_{18}} e_5, \quad e_4 \xrightarrow{\boxed{items[i]=null}_{19}} e_5, \quad e_5 \xrightarrow{\boxed{++i}_{20}} e_6$$

Figure 3. Trace values and reachability relation between environments for removeCheaperThan() when $ab(\boxed{i=0}_{16})$ is assumed

1. For each infeasible environment ϵ, remember the position in the trace where ϵ was first derived.
2. Find all the relations involved in deriving ϵ, and associate the corresponding components with ϵ.
3. Recursively add all components one of the previously selected components is control dependent upon. Also add all components that —when in a different mode— could possibly influence the value of one of the inputs of the components found in the previous step to the set. Note that this set heavily depends on the available component modes, as in the most general modes, almost every value in an environment could be affected, which results in large conflicts and many diagnoses. Therefore, the modes available for each component need to be restricted, relaxing modes only if no suitable diagnoses can be found or complementary models indicate a fault not covered by the stronger modes.

We omit the detailed discussion of cyclic traces for brevity. There, caching prevents multiple analysis of trace parts and ensures linear complexity in the size of the trace

Example 5 *In Example 4, the initial conflict is derived in line 18 in the fourth iteration of the loop. As this is the only branch, the chain of computation is followed backward to the beginning of the trace. The resulting component set is $\{\boxed{i=0}_9, \boxed{i++}_9, \boxed{items[i]=item}_{11}, \boxed{i=0}_{16}, \boxed{items[i].value<value}_{18}, \boxed{++i}_{20}\}$. For example, $\boxed{i=0}_9$ is included because the test in line 18 in the fourth iteration depends on the increment statement in line 20, which transitively depends on the initial value of i. Furthermore, control dependencies lead to adding the components $\boxed{i<items.length}_9, \boxed{items[i]==null}_{10}, \boxed{return}_{12}$, and $\boxed{i<items.length}_{17}$. Finally, $\boxed{items[i]=null}_{19}$ is included in the conflict set because if the component is abnormal, the value and the index expression could be different, thus avoiding the exception in a subsequent iteration. In this example, the components in the conflict are essentially the components in a dynamic slice [14], as only forward computation is used. In more complex situations where backward analysis is used, this is not necessarily the case.*

5 MODEL REFINEMENT

This section investigates an approach to refine a program's model based on results of a preceding abstract interpretation analysis. The basic idea is to eliminate the widening operators from abstract interpretation. To do this, we must prove that the loop always terminates, which can be done for many commonly used classes of loops using different means depending on their structure:

- Syntax-based methods for identifying induction variables can be very useful for loops containing indexed array access elements and handle large classes of practically occurring loops.
- For loops over object structures, we introduce auxiliary variables tracking the number of objects reachable from the loop variable(s). If this number is monotonous and the underlying data structure is acyclic (considering only fields involved in the computation of the loop variables), it is easy to show that the loop is bounded.
- Meta observations, such as *"Every element of the array/data structure must be traversed exactly/at least once"*, also constrain the number of iterations (if the the data structure's size is known).

Example 6 *Assume the current mode assignment for the program in Figure 2 is $\{ab(\boxed{i=0}_{16})\}$. The environment after line 16 is the same as in Example 4. Because the component representing i++ is normal, and i is not updated otherwise, we derive that i is a monotonically increasing loop variable. As the array access expressions implicitly constrain i to $[0..4]$, we derive that the loop is terminating. Backward analysis derives that i must be in $[0..4]$ in case the loop is entered. The number of iterations is therefore bounded by five.*

Once termination has been shown, the loop is expanded into a sequence of traces, where each trace is specialized to handle a subset of the input environment of the original loop. The end of each trace is connected to the beginning of the trace modeling the following iterations. The exit point of the final trace is connected to the program point after the loop. In case the initial values of the variables involved in the loop condition are not known, a connection between the loop entry and the individual traces is added. The widening operator has been replaced by a join operator at the entry of each trace, eliminating the main source of inaccuracy in the model.

The unrolled model of the loop can subsequently be used to eliminate certain transitions from the case statements and thus derive inconsistencies which could not be derived using the original model.

If the number of iterations is small,[6] this model is analyzed immediately. Otherwise, the data flow through the loop is exploited to reduce computational complexity. We distinguish two cases:

- The values of a following iteration possibly depend on values (except loop variables) computed in a previous iteration. In this case, we model k iterations (up to a user-specified threshold) exactly, followed by a model simulating the remaining iterations. This ensures that the information loss in the first few iterations is as small as possible, to provide accurate inputs to subsequent iterations.
- The computations in any two iterations are independent. Here, all iterations are divided into k partitions, which are then analyzed separately and concatenated as before. The partitioning strategy is crucial for the success of this approach. The iterations with most similar input environments should be analyzed together.

If there is only one loop variable, summary information about the values accessed through the loop variable is collected in a preliminary step during parsing of the source code. This results in a set of regular expressions, describing the data access paths parameterized with the loop variables. The summary is used to extract the numerical data that is accessed in each iteration. All accessed values are merged together to create a single abstract value that is used in the algorithm in Figure 4 to group the iterations (D_{ind} denotes the domain of the loop variable).

The algorithm starts with each iteration in a separate partition, and merges the two most similar partitions into one. Similarity between partitions is defined through the ratio between the covered abstract domain values in the merged partition and the sum of the covered values in the individual partitions.

If there are multiple loop variables, we simply subdivide the domain of each of the n variables into $\left\lfloor \sqrt[n]{k} \right\rfloor$ equal-sized regions.

---

[6] Currently, decided by a user-specified threshold, but this could be determined by the size of the model of the loop body, the time estimated to construct the model, the time available for diagnosis, etc.

$$P \leftarrow D_{ind}$$
while $\|P\| \geq 2 \wedge \|P\| > k$ **do**
    pick $\langle p_1, p_2 \rangle \leftarrow \{\langle a, b \rangle \mid \forall_{p,q \in P} : \delta(a,b) \leq \delta(p,q)\}$
    $P \leftarrow P \setminus \{p \in P \mid p \sqsubset p_1 \sqcup p_2\}$
    $P \leftarrow P \cup (p_1 \sqcup p_2)$
end while
return P

$$\delta(a,b) := \frac{covered\_domain(a) + covered\_domain(b)}{covered\_domain(a \sqcup b)}$$

Figure 4. Iteration Partitioning Algorithm

Example 7 *Continuing Example 6, we first perform a preliminary data flow analysis and find that the updates of the array in different iterations are independent (ignoring the dependence through i). Therefore, the summary access paths are used to extract an approximation of the values accessed in each iteration. Here, the only access paths used are* items[i] *and* items[i].value*. This leads to the accessed values* $[10, 10]$, $[20, 20]$, $[25, 25]$, $\perp$, *and* $\perp$, *respectively, for iterations 0 to 4. Assume the number of maximum loop iterations has been set to three. Using the algorithm above, this leads to the following partitioning for i:* $[0, 0], [1, 2], [3, 4]$, *which access the values* $[10, 10], [20, 25]$, *and* $\perp$, *respectively.*

Therefore, three concatenated traces are created and paths from the start of the loop to each of them are inserted.

Analyzing this model using the approach presented in Section 4 leads to the infeasible environment at the loop exit point, because the last partition causes an uncaught NullPointerException *in line 18. The exception is certain as all values accessed in the last two iterations are* null*. Further analysis propagates the contradiction to the program exit, at which point a conflict is detected. Therefore,* $ab(\boxed{\text{i=0}}_{16})$ *cannot be a single fault diagnosis.*

Continuing the analysis until all single fault candidates have been found leads to the final set of diagnoses: $\{ \{ab(\boxed{\text{i=0}}_9)\}, \{ab(\boxed{\text{i++}}_9)\}, \{ab(\boxed{\text{items[i].value<value}}_{18})\}, \{ab(\boxed{\text{items[i]=null}}_{19})\}, \{ab(\boxed{\text{++i}}_{20})\}\}.$

6 RELATED WORK

Automated debugging has been an active area of research for several decades, resulting in a large number of approaches and algorithms.

In Program Slicing [15, 14], statements that cannot influence the value of a variable at a given program point are eliminated by considering the dependencies between the statements. Backward reasoning from output values, as in our approach, is not possible.

[3] introduces an algorithm that compares a faulty program to a close correct variant to determine changes that cause the misbehavior. Although the algorithm seems to be highly effective for test case minimization, it is not clear whether the approach is effectively applicable at a fine grained level, such as basic blocks or statements.

Earlier dependency-based models for Java programs [8] produce relatively coarse results. Value-based models improved on this [9, 10] but still are less expressive than the abstract interpretation based approach when the behavior of components is only partially known.

Abstract interpretation to analyze programs was first introduced by [6], and later extended by [2] to include assertions for abstract debugging. Our framework is strongly inspired by this work, but provides more insight on how to choose approximation operators for debugging, in particular in the case where test information is known.

Recently, model checking approaches have been extended to attempt fault localization in counterexample traces. [1] extended a model checking algorithm that is able to pinpoint transitions in traces responsible for a faulty behavior. [7] presents another approach, which explores the neighborhood of counterexamples to determine causes of faulty behavior. These techniques mostly consider deviations in control flow and ignore data dependencies.

7 CURRENT STATUS & CONCLUSION

We have presented an automatic debugging approach utilizing model-based diagnosis together with an abstract interpretation based conflict detection framework. This framework is able to detect programming errors given a set of test cases, possibly enhanced with partial specifications of the program behavior, e.g., assertions. The abstract interpretation framework allows for more accurate conflict detection than previous models when dealing with loops and absent information. Also, custom abstractions tailored to specific application domains can be integrated easily ([5] describes a model checking based abstraction framework). We showed how results of a previous abstract interpretation based analysis can lead to refined model, resulting in fewer spurious diagnoses.

At the time of writing, the debugging tool is under development and not all steps are carried out automatically. Future work includes the completion of the tool and the evaluation of our debugging strategies and models using a set of larger programs. So far, several small programs (up to approx. 50 lines, excluding comments and test drivers) that were identified as "hard cases" resulting in poor diagnoses when diagnosed using the previous models [8, 9] have been used to develop and evaluate the new model. More detailed evaluation is postponed until our prototype is completed and ready to handle interesting problems automatically.

Current and future research issues include the tight integration of different types of complementary models, such as specifications of valid method call sequences, into the current reasoning framework, as well as further refinement of focus selection and hierarchical modeling approaches. The handling of abnormal loop variable update expressions is also an open topic where the current models do not perform satisfactorily and need to be extended.

REFERENCES

[1] Thomas Ball, Mayur Naik, and Sriram K. Rajamani, 'From symptom to cause: Localizing errors in counterexample traces', in *POPL*, (2003).
[2] François Bourdoncle, 'Abstract debugging of higher-order imperative languages', in *Proc. SIGPLAN Conf. PLDI*, pp. 46–55, (1993).
[3] Holger Cleve and Andreas Zeller, 'Finding failure causes through automated testing', in *Proc. AADEBUG '00 Workshop*, Munich, (2000).
[4] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré, 'Model-based diagnosis meets error diagnosis in logic programs', in *Proc. 13th IJCAI*, pp. 1494–1499, Chambery, (August 1993).
[5] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng, 'Bandera: Extracting finite-state models from Java source code', in *Proc.ICSE-00*, (2000).
[6] Patrick Cousot and Radhia Cousot, 'Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints', in *POPL'77*, pp. 238–252, Los Angeles, (1977).
[7] Alex Groce and Willem Visser, 'What went wrong: Explaining counterexamples', in *SPIN Workshop on Model Checking of Software*, (2003).
[8] Cristinel Mateis, Markus Stumptner, and Franz Wotawa, 'Modeling Java Programs for Diagnosis', in *Proc. ECAI*, Berlin, (August 2000).
[9] Cristinel Mateis, Markus Stumptner, and Franz Wotawa, 'A Value-Based Diagnosis Model for Java Programs', in *Proc. DX'00 Workshop*, Morelia, Mexico, (June 2000).
[10] Wolfgang Mayer and Markus Stumptner, 'Debugging program exceptions', in *Proc. DX'03 Workshop*, pp. 119–124, Washington, D.C., (June 2003).
[11] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers, 'Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization', in *Proc. ACM SIGPLAN-SIGSOFT Workshop PASTE'01*, pp. 66–72, (1991).
[12] Raymond Reiter, 'A theory of diagnosis from first principles', *Artificial Intelligence*, **32**(1), 57–95, (1987).
[13] Markus Stumptner, 'Using design information to identify structural software faults', in *Proc. 14th Australian Joint Conf. on AI*, Springer LNAI 2256, pp. 473–486, Adelaide, (December 2001).
[14] Frank Tip, 'A Survey of Program Slicing Techniques', *Journal of Programming Languages*, **3**(3), 121–189, (September 1995).
[15] Mark Weiser, 'Program slicing', *IEEE TSE*, **10**(4), 352–357, (July 1984).