

Bridging the gap between slicing and model-based diagnosis*

Franz Wotawa
Technische Universität Graz
Institute for Software Technology
8010 Graz, Inffeldgasse 16b/2, Austria
wotawa@ist.tugraz.at

Abstract

Fault localization is considered an important and difficult task in the software engineering process. In the last decades several approaches to fault localization have been published. Some of them are based on either static or dynamic program slicing. In this paper, we present an approach that combines program slicing with the computation of hitting sets. Hitting sets are used in model-based diagnosis to compute diagnoses from conflicting assumptions. We introduce the underlying definitions and algorithms of the approach, and show that the combination of slicing and hitting set computation reduces the number of statements to be considered. The presented approach does not rely on a specific slicing methodology and can be used in combination with static or dynamic slicing.

1. Introduction

Debugging which comprises the activities fault detection, localization, and repair has been an active research area for the past decades. Although most of the research activities can be classified as activities regarding fault detection like formal verification or testing, some effort has been spent in providing tools for fault localization and even less for repair. In this paper, we focus on fault localization and present an approach that combines slicing techniques with the computation of hitting sets, a technique that originates from model-based diagnosis.

Program slicing was introduced by Mark Weiser [15, 16]. He argued that programmers use data-flow and control-flow dependences, and finally slices in order to focus their attention to the more important statements within the pro-

gram in case of incorrect outputs. Mark Weiser took this observation and introduced the concept of static program slices. In [15, 16] a slice is a program where zero or more statements are removed, and which behaves in the same way as the original program for the specified variables at a given location in the program. This definition can hardly be directly implemented because it requires checking program equivalence. Hence, Weiser introduced an approximation algorithm for computing slices in a static way, i.e., only considering the program source code and no dynamic information. Because of the static analysis the Weiser-style slices tend to be larger than necessary for a failure revealing test case.

In order to make slices as small as possible but without losing precision, several improvements have been reported. Some include the use of information about correct program runs. One example is program dicing where the set difference between a static slice for a failure-revealing test-case and the static slice for a program run leading correct outputs is computed. Shahmehri et al. [13] pointed out that dicing is only correct with respect to some very restrictive assumptions. Other improvements and extensions for static slicing have been introduced because of the integration of programming language constructs like procedure calls or concurrency. Horwitz et al. [6] introduced an algorithm for computing the system dependence graph that is an extension of the program dependence graph [2] where procedure calls can be represented. Static slices can be easily computed from such graphs via graph traversal. Krinke [10] improved slicing of programs with procedure calls and extended slicing to handle concurrent programs. Although, those improvements lead to more precise static slices for general programming languages, the use of static slices for debugging is still limited because of their size.

To overcome this problem the concept of dynamic slicing was introduced by Korel et al. [9]. Dynamic slicing additionally takes care of the program execution and, therefore, usually results in smaller slices. But unfortunately dynamic program slices might not include the faulty state-

*The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

```

1.  main(int argc, char *argv[])
2.  {
3.      int red, green, blue, yellow;
4.      int sweet, sour, salty, bitter;
5.      int i;
6.
7.      red = atoi(argv[1]);
8.      blue = atoi(argv[2]);
9.      green = atoi(argv[3]);
10.     yellow = atoi(argv[4]);
11.
12.     red = 2*red; //Error:red = 5*red;
13.     sweet = red*green;
14.     sour = 0;
15.     i = 0;
16.     while (i < red) {
17.         sour = sour + green;
18.         i = i + 1;
19.     }
20.     salty = blue + yellow;
21.     yellow = sour + 1;
22.     bitter = yellow + green;
23.
24.     printf("%d %d %d %d \n",
25.           bitter, sweet, sour, salty);
26.     return 0;
27. }

```

Figure 1. An example program taken from [5]

ments and several extensions like Critical Slicing [1], which combines program mutations and dynamic slicing, Relevant Slicing [19], which introduces a potentially depends relation for the same purpose, and Failure-inducing chops [5], which combine delta debugging [18] with slicing, have been reported. Again, precision and improving the size of slices have been the major driving force of this research. For an overview on slicing we refer the reader to Kamkar [7] or Tip [14]. Other applications of slicing to program debugging include work by Kamkar [8]. Most recently Kusumoto et al. [11] reported on the usefulness of slicing for debugging. In the paper the authors present an empirical study, which shows that programmers are more effective when using slices for debugging.

In this paper, we continue research on improving slicing for debugging. The presented approach potentially leads to smaller slices and is not restricted to a specific slicing methodology. Hence, every technique for computing slices can be used. Before formalizing the approach, we first give an example. The program we are using is given in Figure 1 and was used by Gupta et al. [5] in their paper. The input [1, 5, 8, 2] forms a failure-revealing test case. The out-

puts bitter, sweet, and sour at line 24 are incorrect. For each incorrect output at line 24, we compute a dynamic slice:

- bitter: {7,9,12,14,15,16,17,18,21,22,24}
- sweet: {7,9,12,13,24}
- sour: {7,9,12,14,15,16,17,18,24}

Informally, the semantics of a slice for a slicing criterion comprising a variable at a position and a test case can be stated as the set of statements where each statement contributes to the incorrect computation. In this case we know that at least one of the statements of the slice is responsible for the observed behavior. This observation is equivalent to the following sentence: It is a contradiction to assume that all statements of a slice are correct. Such contradictions are also called conflicts in model-based diagnosis [12].

In order to eliminate all conflicts, we have to take one necessarily not different element from each conflict, i.e., slice, and assume that the element, i.e., statement, behaves not correct. When doing so we eliminate all possible conflicting assumptions. The selected elements have an intersection with every slice and are called hitting sets. In model-based diagnosis the hitting sets of conflicts are diagnoses. In most cases someone is interested in small diagnoses with respect to their size. For our example, we obtain 4 diagnoses of size 1, i.e., 7, 9, 12, and 24, because these statements are elements of each slice. If we are only interested in single faults, then computing the intersection of all conflicts would be sufficient. However, in case of multiple faults the intersection of conflicts can be empty. Therefore, a general approach cannot rely on intersection.

The paper is organized as follows. In the next section we start with the basic definitions and give an algorithm for computing hitting sets. Afterwards, we introduce the approach and present an algorithm which combines hitting sets with slices. We further present a small case study and an extension that handles the closer integration of slicing and hitting set computation, and finally we conclude the paper.

2. Hitting sets

Model-based diagnosis [12] is a troubleshooting methodology, which allows computing explanations for a certainly detected misbehavior directly from the model. Explanations are called diagnoses. Diagnoses are computed from conflicts, i.e., parts of the model, which lead to an inconsistency considering the given observations. For this purpose we use hitting sets, which we define in this section of the paper in order to be self contained. For the formal definitions and an algorithm we refer the reader to Reiter's

work [12]. Greiner et al. [4] presented a corrected version of Reiter’s algorithm.

Hitting sets are defined over a set of sets with the property that the intersection of a hitting set with every given set is not empty.

Definition 1 (Hitting set) A set $\Delta \subseteq \bigcup_{x \in F} x$ for a set of sets F is a hitting set iff the intersection of Δ with all elements of F is not empty, i.e., $\forall x \in F : \Delta \cap x \neq \emptyset$.

A hitting set is minimal if no proper subset of a hitting set is itself a hitting set. Usually we are only interested in minimal hitting sets and if not otherwise mentioned we always refer to minimal hitting sets when using the term hitting set. Note that the definition of minimal hitting set is not based on cardinality. For example the set $\{12\}$ is a minimal hitting set for $\{\{12,13,15\}, \{12,14,16\}, \{12,14\}\}$ but also $\{13,14\}$.

Reiter [12] introduced an algorithm for computing hitting sets that has been improved later by Greiner et al. [4]. The algorithm uses the given sets of sets F and constructs a directed acyclic graph (DAG) in a breadth first manner. After the construction of the DAG the minimal hitting sets correspond to some vertices of the DAG which are labeled with a \checkmark . The algorithm needs not to compute all possible hitting sets. Instead the user can specify the maximum cardinality of the obtained hitting sets. For practical applications especially in cases where the size of the input is large, such a boundary value is of great use. The following algorithm is a variant of the original algorithm where we eliminated one pruning rule. This elimination is possible when assuming that F is a sorted collection with respect to the cardinality of the sets where the left-most element has the smallest one.

Algorithm Hitting-Set-Computation

Input: A sorted collection F of sets with respect to cardinality. The smallest set is assumed to be the left-most element of F . A number $MAX > 0$ which specifies the maximum size of the generated hitting sets.

Output: All minimal hitting sets of F .

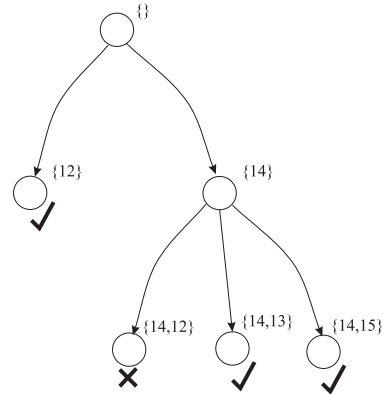
1. Let H be the growing DAG and L be the empty set. Generate a new node n , which is the root node of H , add it to H , let $label(n)$ and $h(n)$ be the empty set. Add n to L , let L' be the empty set and set $i = 0$.
 2. For all nodes n in L do:
 - (a) From left to right search for a set $C \in F$ such that $C \cap h(n)$ is the empty set. If there is no such set, a new minimal hitting set has been found and let $label(n) = \checkmark$.
 - (b) Otherwise, for each $x \in C$ do:
 - i. If there exists a previously handled node m with $h(m) = h(n) \cup x$, then generate a new arc from n to m .
 - ii. Otherwise, generate a new node n' with $h(n') = h(n) \cup x$, and an arc from n to n' . If there exists a previously handled node m with $label(m) = \checkmark$, and $h(m) \subset h(n')$, then close node n' and let $label(n') = \times$. Otherwise, add n' to L' .
 - (c) Let $i = i + 1$.
3. If L' is not empty, and $i \leq MAX$, let L be L' and $L' = \emptyset$, and go to 2.
4. Otherwise, return a set comprising $h(n)$ for all nodes n with $label(n) = \checkmark$.

- i. If there exists a previously handled node m with $h(m) = h(n) \cup x$, then generate a new arc from n to m .
- ii. Otherwise, generate a new node n' with $h(n') = h(n) \cup x$, and an arc from n to n' . If there exists a previously handled node m with $label(m) = \checkmark$, and $h(m) \subset h(n')$, then close node n' and let $label(n') = \times$. Otherwise, add n' to L' .

- (c) Let $i = i + 1$.
3. If L' is not empty, and $i \leq MAX$, let L be L' and $L' = \emptyset$, and go to 2.
4. Otherwise, return a set comprising $h(n)$ for all nodes n with $label(n) = \checkmark$.

The hitting set algorithm obviously terminates for every finite set F and MAX . If F is not empty, then the algorithm has at least two iterations. All hitting sets can be computed by setting MAX to the number of elements stored in F 's set, i.e., $MAX = |\bigcup_{x \in F} x|$.

The hitting set DAG for $F = \{\{12,13,15\}, \{12,14,16\}, \{12,14\}\}$ is given as follows where the values of h for each node are given under parentheses ($\{\}$):



Note that the algorithm has to be called on the sorted collection $F' = \{\{12, 14\}, \{12, 14, 16\}, \{12, 13, 15\}\}$ and not on the original set F . Hence, we finally obtain 3 minimal hitting sets.

3. Hitting sets and slices

In this section, we formally introduce our fault localization process. We assume a program Π that is written in a programming language \mathcal{L} . We further assume a semantics of \mathcal{L} defined by a function $\llbracket \cdot \rrbracket : \mathcal{L} \times \Sigma \mapsto \Sigma$ which maps programs and states to new states. In this definition a state $s \in \Sigma$ specifies values for variables used in the program.

We further assume that the program Π is correct with respect to the grammar of \mathcal{L} and halts on every given input. A test case is a tuple (I, O) where $I \in \Sigma$ is the input and $O \in \Sigma$ is the expected output. A program Π passes a test case $t = (I, O)$ iff $\llbracket \Pi \rrbracket I \supseteq O$. Otherwise, we say that the program fails. Because of the use of the \supseteq operator also partial test-cases are allowed which do not specify values for all output variables. If a program passes a test case t , then t is called a positive test case. Otherwise, the test case is said to be a negative test case. Note that we do not consider inconclusive test cases explicitly. In cases where inconclusive test cases exist, we treat them like being positive test cases. Since we are only considering negative test cases for fault localization this assumption has no influence on the final result. A test suite TS for a program Π is a set of test cases and can be partitioned into two disjoint sets comprising only positive (*POS*) respectively negative (*NEG*) test cases, i.e., $TS = POS \cup NEG \wedge POS \cap NEG = \emptyset$.

For a negative test case $t = (I, O) \in NEG$ we know because of the definition that there must be some variables $CV_t = \{x_1, \dots, x_k\}$ where for all $i \in \{1, \dots, k\}$ $x_i = v_i \in O$ and $x_i = w_i \in \llbracket \Pi \rrbracket I$ follows that $v_i \neq w_i$. We call such variables x_1, \dots, x_k conflicting variables. For each of the variables x_1, \dots, x_k we compute a slice $S(x_1), \dots, S(x_k)$ as follows: $S(x_i) = SLICE(\Pi, \langle t, n, \{x_i\} \rangle)$ where $SLICE$ is a function implementing the computation of either static or dynamic slices, t is a test case, n is the line of the program where variable x_i is known to hold the wrong value. Note that we have no restrictions on the computation of slices but using slicing algorithms that are incorrect or produce imprecise results will cause our approach to compute itself incorrect or imprecise results. The corresponding conflict set for a negative test case t is now given as $C_t = \{S(x) | x \in CV_t\}$. From this conflict set we compute all minimal diagnosis, e.g., $DIAGS_t = \{\Delta | \Delta \in HS(C_t)\}$ where HS implements the introduced hitting set algorithms for the given set C_t .

Before discussing some implications of the above definitions we illustrate the approach using our example program from Fig. 1. Given a test cases that is described in the introduction we obtain the following sorted collection of conflicts when using dynamic slicing: $F = \{\{7,9,12,13,24\}, \{7,9,12,14,15,16,17,18,24\}, \{7,9,12,14,15,16,17,18,21,22,24\}\}$. From this collection our implementation of the hitting set algorithms computes 9 diagnoses within a fraction of a second: $\{7\}, \{9\}, \{12\}, \{24\}$ are single fault diagnoses and $\{13,14\}, \{13,15\}, \{13,16\}, \{13,17\}, \{13,18\}$ are double fault diagnoses.

When having a look at the obtained theory and results someone might ask why not using the intersection of conflicts directly instead of computing minimal diagnoses using a hitting set algorithm? To answer this question, we

first define the intersection formally as $INTERSECT_t = \{\{n\} | n \in \bigcap_{x \in C_t} x\}$. From this definition follows that for every set $\{i\} \in INTERSECT_t$, i itself has to be an element of every conflict. Accordingly to the definition of hitting sets and diagnosis, $\{i\}$ would also be element of $DIAGS_t$. Hence, all elements of $INTERSECT_t$ are single fault diagnosis of $DIAGS_t$. But in cases where $INTERSECT_t$ is empty, $DIAGS_t$ still provide usefull information, e.g., that there are no single fault diagnoses. The same results cannot be obtained when using the intersection operator. Therefore, we conclude that the hitting set computation is more general than computing the intersection only. The following corollary summarizes these findings.

Corollary 1 *Given a program Π and a negative test case t . The intersection $INTERSECT_t$ of all conflict sets for t given Π is a subset of the set of all diagnosis $DIAGS_t$ for the same conflict set. If $INTERSECT_t$ is the empty set, then all elements of $DIAGS_t$ have a size greater than 1, i.e., in this case there are only multiple fault explanations for a negative test case.*

Our diagnosis approach only delivers better results when there are more different slices. In cases where only one slice is available because only one output contradicts the expected output values of a test case, every element of the slice is a minimal single fault diagnosis. Hence, the approach does not gain new information in this case.

Corollary 2 *Given a program Π and a negative test case t . If the set of contradicting variables contains only one element x , then all elements of the corresponding slice $SLICE(\Pi, \langle ll(\Pi), \{x\} \rangle)$ are single fault diagnosis.*

Note that the above process makes only use of a single negative test case. It is of course also possible to include all slices coming from all negative test cases for a particular program comprising the same input and output variables. In this case depending on the construction of the test cases multiple faults becomes more likely.

4. Case study

In order to further evaluate the capabilities of the presented approach, we started with a case study. This study includes 5 small programs ranging from 12 to 129 lines of code. All of the used examples have several inputs and several outputs. All programs except the first one are implementing digital circuits. The first one is the one from Fig. 1. Faults were introduced in the program manually. Test cases were computed randomly using the original programs as specifications. During random test case generation a lot of failure-revealing test cases, which make more than

1 output incorrect, could be obtained. From the obtained slices we computed the diagnoses using a Java implementation of the introduced hitting set algorithm. In all cases the diagnoses could be obtained within less than a second on a standard PC.

The statistical information regarding the considered programs as well as the obtained results are given in Table 1. `example` is the program from Fig. 1. `alu` is a Java implementation of an arithmetic logic unit. `4_bit_adder` is a Java implementation of a binary 4 bit adder. `c17` is a Java implementation of a ISCAS85 circuit. The ISCAS85 suite is used as benchmark suite in the hardware design community. Finally, `code_converter` implements a seven segment code conversion device.

Table 1 gives the lines of code (LOC), the number of slices, the minimum (Min) and maximum (Max) size of the slices, the size of the union of all obtained slices (All), the number of single fault diagnoses (Bugs), where the set of bugs is the result of the hitting set computation, and the percentage of single fault diagnoses with respect to the lines of code. In all examples the number of single fault diagnoses is smaller than the smallest slice. If we build the union of the slices for each incorrect output, the reduction would be about 50 percent. When using the introduced approach, the reduction is between 80 to 95 percent. This means that in the best case only the remaining 5 percent of the source code has to be considered for further investigation during debugging.

5. Extensions

It has been shown in various papers, e.g., [15, 16] and [11], that programmers effectively make use of slices during fault localization. Although, the introduced notation of diagnosis lead to improvements in terms of a reduction of statements to be considered, it might not work as expected in all situations. Consider for example the case where only multiple fault diagnoses are available. A programmer might gain important information from the diagnoses. But the listed diagnoses do not allow for providing an overview. Only diagnosis after diagnosis can be looked at during the whole debugging process. If considering our example program in Fig. 1, we have to analyze the 5 double fault diagnoses one by one.

In order to overcome this problem, we describe how to map back diagnoses to some sort of summary slices. For this purpose, we enhance the information provided by a slice with a probability value that is assigned to each element of the slice. We call such a slice comprising statements and corresponding probabilities a HS-slice.

Let $DIAGS_t$ be the set of diagnosis obtained from a set of slices F for a negativ test case t , and a program Π using the hitting set algorithm. For each diagnosis $\Delta \in DIAGS_t$

we compute its probability. This probability is equivalent to the probability of the state that all elements in Δ are incorrect and that all other statements are correct. Formally, this probability is stated as follows (when assuming independence of failure):

$$p(\Delta) = \prod_{s \in \Delta} p_F(s) \cdot \prod_{s' \in \Pi \setminus \Delta} (1 - p_F(s'))$$

The fault probability of a statement s , i.e., $p_F(s)$, is usually not given. In such cases the assumption that all statements fail with equal probability is used. Using this assumptions the fault probability of statement s becomes $p_F(s) = 1/|\Pi|$, where $|\Pi|$ denotes the number of statements of program Π . Because the same probability applies for all statements, we drop $p_F(s)$ and use p_F from here on instead. We finally obtain the fault probability of a diagnosis Δ :

$$p(\Delta) = p_F^{|\Delta|} \cdot (1 - p_F)^{|\Pi \setminus \Delta|}$$

The probability of a statement s can be obtained by computing the sum of the fault probabilities of the diagnoses where the statement is an element.

$$p(s) = \sum_{\Delta \in DIAGS_t \wedge s \in \Delta} p(\Delta)$$

We now have all necessary pieces to define HS-slices.

Definition 2 (HS-slice) *Given a program Π , a test case t . A HS-slice S is a set of pairs that is obtained from the set of diagnoses $DIAGS_t$ as follows:*

$$S = \{(s, p(s)) \mid \exists \Delta \in DIAGS_t : s \in \Delta\}$$

Using this definition we obtain the following HS-slice for the example program from Fig. 1:

$$\{(7,0.0139), (9,0.0139), (12,0.0139), (13,0.0027), (14,0.0005), (15,0.0005), (16,0.0005), (17,0.0005), (18,0.0005), (24,0.0139)\}$$

Note that this slice is also smaller than the union of the three slices. It specifically indicates the statements with the highest fault probability. The fact that every double fault diagnoses has to have statement 13 as an element is also represented in an appropriate way.

6. Conclusion

The application of model-based diagnosis to software debugging is not new. Friedrich and colleagues [3] used model-based diagnosis together with a dependence-based model to localize bugs in VHDL programs. Wotawa [17] proved that the strong relationship between static slicing and model-based debugging using dependence-based models. In this paper, we present a more general approach for the integration of model-based debugging and slicing. We presented a first case study and finally an extension which allows for an easy integration with existing slicing-based approaches.

Because the approach is based on existing slicing techniques, the overall outcome depends on those techniques.

Table 1. Diagnosis results

Program	LOC	Inputs	Outputs	No of Max	Size of Slices			Bugs	Percentage
		Slices	Min		All				
example	4	4	26	3	5	11	12	4	15.4
alu	14	8	129	4	32	57	65	25	19.4
4_bit_adder	8	5	56	4	6	13	25	3	5.4
c17	5	2	12	2	3	5	6	2	16.7
code_converter	6	7	68	5	10	13	30	7	10.3

Limitations of used slicing techniques will also be limitations of the approach. In cases where only one output is incorrect and, therefore, where only one slice is available, the approach does not improve the final outcome. This is not a severe problem because the computational requirements are not very demanding especially in the case of only one slice.

References

- [1] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 121–134, 1996.
- [2] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
- [3] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.
- [4] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter’s theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.
- [5] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Automated Software Engineering (ASE)*, pages 263–272, November 2005.
- [6] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependency Graphs. In *Proceedings of the SIGPLAN’88 Conference on Programming Language Design and Implementation*, pages 35–46, Atlanta, Georgia, 1988.
- [7] Mariam Kamkar. An overview and comparative classification of program slicing techniques. *J. Systems Software*, 31:197–214, 1995.
- [8] Mariam Kamkar. Application of program slicing in algorithmic debugging. *Information and Software Technology*, 40:637–645, 1998.
- [9] Bogdan Korel and Janusz Laski. Dynamic Program Slicing. *Information Processing Letters*, 29:155–163, 1988.
- [10] Jens Krinke. Advanced slicing of sequential and concurrent programs. In *20th International Conference on Software Maintenance*. IEEE, 2004.
- [11] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7:49–76, 2002.
- [12] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
- [13] Nahid Shahmehri, Mariam Kamkar, and Peter Fritzson. Usability criteria for automated debugging systems. *J. Systems Software*, 31:55–70, 1995.
- [14] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.
- [15] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.
- [16] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
- [17] Franz Wotawa. On the Relationship between Model-Based Debugging and Program Slicing. *Artificial Intelligence*, 135(1–2):124–143, 2002.
- [18] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), feb 2002.
- [19] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault localization. In *Sixth International Symposium on Automated & Analysis-Driven Debugging (AADEBUG)*, pages 33–42, 2005.