Refining Spectrum-based Fault Localization Rankings*

Rui Abreu† Wolfgang Mayer‡ Markus Stumptner‡ Arjan J.C. van Gemund†

†Embedded Software Lab
Delft University of Technology
The Netherlands
{r.f.abreu, a.j.c.vangemund}@tudelft.nl

‡Advanced Computing Research Centre
University of South Australia
Australia
{mayer, mst}@cs.unisa.edu.au

ABSTRACT

Spectrum-based fault localization is a statistical technique that aims at helping software developers to find faults quickly by analyzing abstractions of program traces to create a ranking of most probable faulty components (e.g., program statements). Although spectrum-based fault localization has been shown to be effective, its diagnostic accuracy is inherently limited, since the semantics of components are not considered. In particular, components that exhibit identical execution patterns cannot be distinguished. To enhance its diagnostic quality, in this paper, we combine spectrum-based fault localization with a model-based debugging approach based on abstract interpretation within a framework coined DEPUTO. The model-based approach is used to refine the ranking obtained from the spectrum-based method by filtering out those components that do not explain the observed failures when the program's semantics is considered. We show that this combined approach outperforms the individual approaches and other state-of-the-art automated debugging techniques.

Categories and Subject Descriptors

D.2.5 [**Software engineering**]: testing and debugging—*debugging aids, diagnostics*

General Terms

Reliability, Experimentation, Measurement.

Keywords

Program spectra, abstract interpretation, fault localization.

1. INTRODUCTION

Considerable costs are attached to locating and eliminating problems in software systems during development as well as after deployment [15]. Hence, numerous approaches have been proposed to automate parts of the testing and debugging process to help detect more defects earlier in the development cycle and to guide software engineers towards possible faults.

(Semi-)Automated fault localization techniques can be classified as *statistics-* or *model-based* approaches. By analyzing abstractions of program traces (also called *program spectra*), spectrum-based fault localization (SFL) and other statistics-based approaches yield a list of suspect program components sorted by their likelihood to be at fault. As components involved in at least one failed computation will be part of this ranking, the produced output can be very large and many components may need to be inspected in some cases. Although efficient in locating faulty components, SFL is unable to distinguish between components that exhibit the same execution pattern or that appear in all computations. Since this technique is efficient in practice, this and other *dynamic analysis* techniques are attractive for large modern software systems [19].

Statistical techniques are rather dependent on the availability of a suitable test suite. Better results can often be achieved if a model of the correct program behavior is available. Model-based software debugging (MBSD) techniques have been advocated as powerful debugging aid that isolate faults in complex programs [11]. By comparing the state and behavior of a program to what is anticipated by its programmer, model-based reasoning techniques separate those parts of a program that may contain a fault from those that cannot be responsible for observed symptoms. Although competitive with other state of the art automated debugging approaches [11], MBSD is computationally much more demanding than SFL and may still produce a large output that lacks ranking information.

In a previous work [10], SFL has been integrated within MBSD to prioritize the diagnostic report given by the latter in order to guide the search. The authors conclude that the combination of the two approaches reduces the effort to track down faulty components. However, the computational complexity of this approach is determined by MBSD. As a result of its high computational complexity, the approach proves prohibitive for large programs.

In this paper, we present a new framework, coined DEPUTO[1], that integrates SFL with MBSD to focus the search by filtering ranked results. Our approach first uses SFL to compute the ranked list of likely faulty components, and, subsequently, applies MBSD to refine the ranking by removing components that do not explain observed failures. Our algorithm inherits the low computational complexity from SFL and the significantly improved diagnostic accuracy from MBSD. While the underlying principle is similar to a previous work [10], we achieved an 2.5 times (on average) speed-up while maintaining the accuracy of the previous approach.

The paper is organized as follows. The principles of spectrum-

*This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the BSIK03021 program.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

[1]Latin for pruning.

based fault localization are outlined in Section 2, followed by a discussion of model-based debugging in Section 3. The combined framework is discussed in Section 4. Empirical validation of our approach and our findings are given in Section 5. Section 6 discusses relevant related works, followed by the conclusion.

2. SFL

Spectrum-based fault localization (SFL) is a dynamic program analysis techniques that has shown that comparing the program behavior over multiple test runs can indicate which program components may be likely to contribute to an observed program failure.

In the following, we assume that a program $\mathcal{P}$ comprises a set of components $\mathcal{C}$ (statements in the context of this paper) and is executed using a set of test cases $\mathcal{T}$ that either pass or fail, with $M = |\mathcal{C}|$ and $N = |\mathcal{T}|$, respectively. Program (component) activity is recorded in terms of program spectra [2, 8]. These data are collected at run-time and typically consist of a number of counters or flags for the different components of a program. We use the so-called *hit spectra* that indicate whether a component was involved in a (test) run or not.

Both spectra and pass/fail information is input to SFL. The combined information is expressed in terms of the $N \times (M+1)$ *participation matrix* O. An element o_{ij} is equal to 1 if component j took part in the execution of test run i, and 0 otherwise. The rightmost column of O, the error vector e, represents the test outcome. The element $e_i = o_{i,m+1}$ is equal to 1 if run i *failed*, and 0 if run i *passed*. For $j \leq M$ and $i \leq N$, the row O_{i*} indicates whether a component was executed in run i, whereas the column O_{*j} indicates in which runs component j was involved.

In SFL one measures the similarity between the error vector e and the activity profile vector O_{*j} for each component j. This similarity is quantified by a *similarity coefficient*, expressed in terms of four counters $a_{pq}(j)$ that count the number of positions in which O_{*j} and e contain correspondent values p and q; that is, for $p, q \in \{0, 1\}$, we define $a_{pq}(j) = |\{i \mid o_{ij} = p \wedge e_i = q\}|$. In this paper, the Ochiai similarity coefficient, known from molecular biology, is used. Previous investigations have identified it as the best coefficient to be used for SFL [2], consistently outperforming several other coefficients, such as the one used in the Tarantula tool [8]. It is defined as

$$s_j = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}}$$

The similarity coefficient s_j associated with each component $C_j \in \mathcal{C}$ indicates the correlation between the executions of C_j and the observed incorrect program behavior. Applying the hypothesis that closely correlated components are more likely to be relevant to an observed misbehavior, s_j can be reinterpreted as "fault probability" and components can be listed in order of likelihood to be at fault.

To illustrate how SFL works, consider the program in Figure 1, which contains a defect in line 9 – instead of assigning 0 to variable i, it assigns 1. An observation for this program consists of program inputs, i.e., values for variables tbl, n and k, together with the anticipated result value returned by the algorithm. For example, the assignments $tbl \leftarrow [90, 21, 15, 0, 0, 0, 8, 23, 0, 0, 0, 0, 50, 60, 59]$, $n \leftarrow 16$, $k \leftarrow 90$ and the assertion $result = 0$ could be an "observation" specifying the inputs and the desired result of a particular program execution. Since the result (-1) obtained by running the program on the given inputs contradicts the anticipated result (0), it shows that the program is incorrect.

Executing the program in Figure 1 using the test case described above results in the first row vector in the participation matrix in

```
function FINDINDEX(tbl, n, k)
    ▷ Find the index of key k in the hash table
      tbl[0, ..., n − 1], or −1 if not found.
      Assumes that tbl contains a free slot.
 1    i ← HASH(k)                    ▷ Hash key
 2    while tbl[i] ≠ 0 do            ▷ Empty slot?
 3        if tbl[i] = k then
 4            return i               ▷ Found match
 5        end if
 6        if i < n − 1 then          ▷ At end?
 7            i ← i + 1              ▷ Try next
 8        else
 9            i ← 1                  ▷ Wrap around (Fault)
10        end if
11    end while
12    return −1                      ▷ Not found
end function
```

Figure 1: Algorithm to search in a hash table

C_1	C_2	C_3	C_4	C_6	C_7	C_9	C_{12}	e
1	1	1	0	1	1	1	1	1
1	1	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	0	0
1	1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	0	1
0.58	0.58	0.63	0.00	0.71	0.71	0.71	0.58	

Figure 2: Participation Matrix

Figure 2. The vector contains a single 0 entry, indicating that all components but C_4 are executed. Since the returned value does not match the anticipated result, the entry in the error vector is set to 1. Assume that further tests are executed to yield the other (5) rows in the participation matrix.

For each component C_j the Ochiai similarity s_j is given below the matrix. For C_3, the similarity coefficient s_3 is 0.63: as can be seen from the third column in the matrix, there are two failing test runs when C_3 is executed ($a_{11}(3) = 2$), no failing run when C_3 does not participate ($a_{01}(3) = 0$), and three successful executions when C_3 is involved ($a_{10}(3) = 3$). C_6, C_7 and C_9 are considered to be most closely correlated with failing tests and should be examined first. Conversely, C_4 is not considered relevant at all.

Since SFL abstracts a program's behavior into a model that is not suitable for reasoning about the semantics of individual components, results may be affected from the following phenomena:

- If a fault lies in a component that participates in all runs (for example, an initialization component), the component is likely not to be ranked high;
- SFL cannot distinguish between components that exhibit identical execution patterns (such as components C_6, C_7, and C_9);
- Nested components executed after the fault is hit are likely to outrank the faulty component. For example, branches of a conditional statement are likely to outrank those components preceding it.
- Many components may be included in the ranking. In our example, seven out of eight components are included in the report.

3. MBSD

Model-based diagnosis has been proven successful in aiding developers in locating the root cause of failures in physical systems by using a model of the systems' intended behavior [13]. For software programs, however, creating such a model can be as difficult and error-prone as building the actual implementation [12]. Model-based debugging [11] aims to close the gap between powerful formal analysis techniques and execution-based strategies in a way that does not require the end-user to possess knowledge of the underlying reasoning mechanisms. Here, an adaptation of the classic "reasoning from first principles" [13] (that is, information directly available from program execution and source code) paradigm borrowed from diagnosis of physical systems is particularly appealing, since much of the complexity of the formal underpinnings of program analysis can be hidden behind an interface that resembles the end-user's traditional view of software development. In contrast to statistics-based approaches, MBSD is less dependent on large test suites as it exploits a model of normal behavior.

Different from classical model-based diagnosis, where a correct model is furnished and compared to symptoms exhibited by an actual faulty physical artifact, debugging software reverses the roles of model and observations. Instead of relying on the user to formally specify the desired program behavior, the (faulty) program is taken as its own model and is compared to examples representing correct and incorrect executions. Hence, the model in MBSD reflects the faults present in the program, while the observations indicate program inputs and correct and incorrect aspects of a program's execution. Observations can either be introduced interactively or can be sourced from existing test suites.

In the following, we briefly outline the model construction. More detailed discussion can be found in [11]. Similarly to SFL, a program is partitioned into components, each representing a particular fragment in the program's source code. The behavior of each component is automatically derived from the effects of individual expressions the component comprises. Connections between components are based on control- and data-dependencies between the program fragments represented by each component.

Assume a model at statement granularity is to be created from the program in Figure 1. For each statement s, a separate component is created that is comprised of the expressions and sub-expressions in s. The inputs and outputs of the components correspond to the used and modified variables, respectively. Connections between the components are created to reflect data dependencies between statements in the program (as determined by a simple data flow analysis). Additional variables and components may be introduced to correctly capture data flow at points where control flow paths may split or merge. The component C_7 corresponding to statement 7 in Figure 1 is represented as a component with input i_2 and output i_7. Here, i_7 represents the result value of statement 7, and i_2 denotes the previous value of variable i that is implicitly defined at the loop head in line 2.

Similar to classical model-based diagnosis, the model also provides different operating modes for each component, where the "correct" (*healthy*) mode h_j of component C_j corresponds to the case where C_j is not to blame for a program's misbehavior. In this case, C_j is defined to function as specified in the program. Conversely, when component j is assumed "not healthy" ($\neg h_j$), C_j may deviate from the program's behavior. For example, the behavior of C_7 can be expressed as the logical sentence

$$h_7 \Rightarrow i_7 = i_2 + 1. \quad (1)$$

In the case where C_7 is considered faulty ($\neg h_7$ is true), the effect on i_7 is left unspecified.

The main difference between the original program and its model is that the model represents the program in a form that is suitable for automated consistency checking and prediction of values in program states in the presence of fault assumptions. This includes program simulation on partially defined program states, using abstract interpretation [5], and backward propagation of values or constraints, which would not occur in a regular (forward) program execution.

Since the resulting model includes the same faults as the program, means to compensate for incorrect structure and behavior of components must be introduced. While heuristics to diagnose structural deficiencies in physical systems can be based on invariants and spatial proximity [4], in software, the model must be adapted and restructured once a defect in its structure has become a likely explanation. Here, detection and model adaptation must be guided by using abstract assertions that capture simple "structural invariants" [11]. Also, since different fault assumptions may alter the control and data flow in a program, models may be created lazily rather than in the initial setup stage.

A trade-off between computational complexity and accuracy can be achieved by selecting different abstractions and models [11], both in terms of model granularity and representation of program states and executed transitions. In Eq. 1 the representation of program state has been left unspecified. Using an interval abstraction to approximate a set of values, sentence (1) becomes a constraint over interval-valued variables i_2 and i_7 [11]. Another possible abstraction is to encode the operation as logical sentences over the variables' bit representations [11]. In this paper, we use the interval abstraction, since it provides good accuracy but avoids the computational complexity of the bit-wise representation.

Similar to consistency-based diagnosis of physical systems [13], from discrepancies between the behavior predicted by the model and the behavior anticipated by the user, sets of fault assumptions are isolated that render the model consistent with the observations. Formally, the MBSD framework is based on extensions to Reiter's consistency-based framework, where a diagnosis is a set of faulty components that together explain all observed failures. Diagnoses are obtained by mapping the implicated components into the program's source code [11].

When this approach is used with the program and the test case from Section 2, a contradiction is detected when the assertion checking the expected result fails. The (cardinality-) minimal fault assumptions that are consistent with our test specification are: $\{\neg h_1\}$, $\{\neg h_7\}$, $\{\neg h_9\}$, and $\{\neg h_{12}\}$. Hence, the statements in lines 1, 7, 9 and 12 are considered the possible root causes of the symptoms. Any other statement alone cannot explain the incorrect result, since the result remains incorrect even if a statement is altered.

Conversely to SFL, the model-based technique captures the semantics of programming constructs, but does not assign ranking information to candidate explanations. Hence, in this respect the techniques complement each other.

4. DEPUTO

In this section we describe how we combine the spectrum-based and the model-based approaches described earlier, in order to capture the best characteristics of both techniques.

Algorithm 1 outlines our combined approach. The algorithm executes in three stages, with the similarity-based approach used in the setup stage (steps 1 to 6), feeding into the subsequent model-based filtering stage (steps 7 to 16), followed by an optional best-first search stage (lines 17 to 24). This combination has significantly lower resource requirements than applying MBSD on the

Algorithm 1 DEPUTO Algorithm

Inputs: Program $\mathcal{P}$, set of test cases $\mathcal{T}$

Output: Fault assumptions explaining failed test runs

```
1  C ← CreateComponents(P)
2  M ← GetComponentMatrix(C, P, T)
3  ⟨T_P, T_F⟩ ← Partition(M, T)
4  R ← SFL(M)                          ▷ Apply SFL
5  S ← ∅                               ▷ Skipped components
6  I ← ∅                               ▷ Inspected components
7  repeat
8      Ĉ ← Ranking_pop(R)
9      D ← MBSD(Ĉ, T_F)                ▷ Apply MBSD
10     I ← I ∪ C
11     if D_bug ∈ D is confirmed faulty then
12         return D_bug
13     else
14         S ← S ∪ (Ĉ \ D)
15     end if
16     R ← R \ Ĉ
17 until R = ∅
18 while S ≠ ∅ do
19     Ĉ ← PDG_Ranking_pop(S, I)
20     I ← I ∪ Ĉ
21     if C_bug ∈ Ĉ is confirmed faulty then
22         return {¬h_bug}
23     end if
24 end while
25 return ∅                            ▷ No explanation found
```

whole program and using SFL only to rank results as proposed in [10]. We start by partitioning the program $\mathcal{P}$ into a set of components $\mathcal{C}$ and execute $\mathcal{P}$ on the available test cases $\mathcal{T}$ to obtain the participation matrix $\mathcal{M}$. Using $\mathcal{M}$, we partition $\mathcal{T}$ into passing tests ($\mathcal{T}_P$) and failing ones ($\mathcal{T}_F$). From $\mathcal{M}$, a sorted list of components $\mathcal{R}$ in order of likelihood to be at fault is obtained as described in Section 2 (line 4).

In the subsequent loop, MBSD (line 9) is used to eliminate the top-ranked candidate explanations that are not considered valid explanations by the model-based approach. Instead of applying MBSD once to compute *all* explanations and present the ranked candidates to the user, an incremental strategy permits to stop early once a fault has been identified. First, the set of components $\widehat{\mathcal{C}}$ with the highest similarity coefficient in $\mathcal{R}$ are obtained using the RANKING_POP($\mathcal{R}$) function.Second, function MBSD($\widehat{\mathcal{C}}, \mathcal{T}_F$) returns a set of candidate explanations $\mathcal{D} \subseteq \widehat{\mathcal{C}}$ that explain observed failures $\mathcal{T}_F$. Finally, if the fault is in the returned set, the algorithm stops; otherwise none of the candidates represent valid explanations and other must be generated. The algorithm stops once no more explanations could be found or if none of the remaining components was executed for a failing test. $\mathcal{S}$ is the set of components that are implicated by SFL but not by MBSD.

If no explanation is found after all components implicated by MBSD have been explored, we employ a best-first search procedure that traverses the program along dependencies between components with decreasing fault similarity. No explanation may be found if the fault affects component inter-dependencies such that the fault assumptions and model abstraction can no longer represent the fault. In line 18, the set of components with maximum fault similarity that are connected to the previously explored components is returned. Function PDG_RANKING_POP($\mathcal{S}, \mathcal{I}$) returns the set of components in S with highest similarity that are directly connected to the previously inspected set of component $\mathcal{I}$. If the component is confirmed to be (part of) a valid explanation, the search stops and the diagnosis is returned. Note that the explanation may only cover part of the true fault. Line 24 in Algorithm 1 can only be reached if the faulty program fragment is not covered by any component, or if the user oracle that decides whether an explanation is indeed a satisfying explanation is imperfect and may miss a fault.

Applying Algorithm 1 using the test suite from the example in Section 2, $\{\neg h_7\}$ and $\{\neg h_9\}$ are obtained as candidate explanations. Both candidates are associated with the highest similarity coefficient 0.71.

Notably, this result improves upon both individual fault localization procedures. Different from pure SFL, $\{\neg h_6\}$ is no longer considered an explanation. Conversely, candidates $\{\neg h_1\}$ and $\{\neg h_{12}\}$ obtained using pure MBSD are low-ranking in SFL and hence omitted at this stage. ($\{\neg h_{12}\}$ is already eliminated by pure MBSD when using the second failing test case in the example.)

Without further information, neither approach can discriminate between the two remaining candidate explanations. Since it is assumed that the user acts as oracle that can reliably recognize true faults, the algorithm stops in the first iteration (in line 12), once the statement in Figure 1 corresponding to $\{\neg h_9\}$ has been confirmed to be incorrect.

5. EMPIRICAL EVALUATION

To gain a better understanding of the combined approach, in this section, we empirically evaluate its efficiency. First, we introduce the program under analysis and the evaluation metric.

5.1 Experimental Setup

Program under analysis In our study we use the *TCAS* program as taken from the *Siemens Test Suite*[2], which is a test bench commonly used in the debugging community. *TCAS* simulates the resolution-advisory component of a collision avoidance system similar to those found in commercial aircraft. It consists of 138 lines of C code and takes twelve parameters as input; the numeric result value encodes one out of three possible resolution advisories. The program comes with 1608 test cases and 41 different variants with known faults. For each variant, an average of forty test cases reveal a fault. In our experiments, all available test cases were used.

Evaluation Metric In the fault diagnosis research community rank- [2, 8, 16] and dependency-based [9, 14] metrics have often been used. The former quantify the quality of a result based on the ranking position of the faulty component relative to all components, and is mainly used with techniques that rank components in a program. In contrast, dependency-based measures typically operate on the program dependence graph (PDG) and are mainly applied to evaluate techniques that either do not rank components (for example MBSD) or do not rank all components of a program (such as SOBER [9]). Essentially, starting with the set of blamed components, dependencies between components are traversed in breadth-first order until the fault has been reached. The quality of a diagnostic report is measured as the fraction of the PDG that is traversed. Both metrics quantify the percentage of a program that needs to be inspected in order to find the fault. We refer to them as SCORE.

To assess the accuracy of DEPUTO, we use both metrics. First, if a fault is found in the refining phase (lines 7 to 16 in algorithm 1),

[2] `http://sir.unl.edu/`

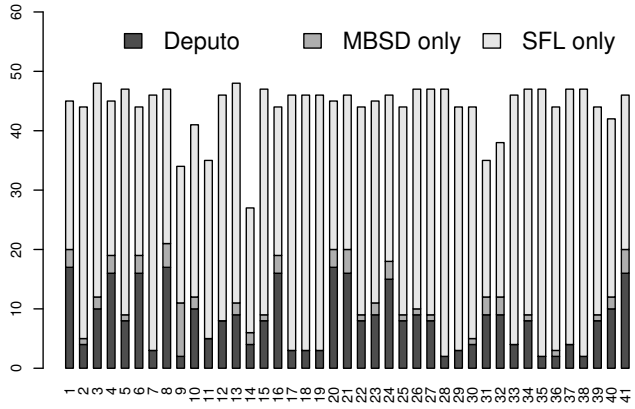

Figure 3: Components implicated by SFL and MBSD

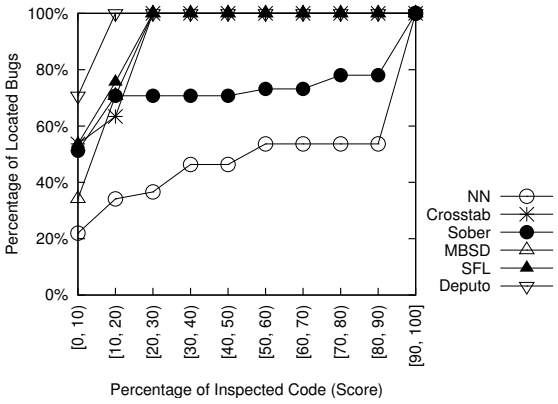

Figure 4: Debugging efficiency

the SCORE is given by

$$\text{SCORE} = \frac{|\mathcal{I}|}{M} \cdot 100\%,$$

where $|\mathcal{I}|$ denotes the number of inspected components. However, if no fault is found in this phase then we use the PDG-based metric by traversing the ranking starting with the previous inspected set of components $\mathcal{I}$ (lines 17 to 23).

5.2 Experimental Results

Results for the individual approaches have already been published elsewhere. MBSD using an interval abstraction requires 11 statements to be inspected on average [11]. The median SCORE is 13% (14% on average) for *TCAS*. Note that when MBSD fails to implicate the faulty component, more statements than just those in the diagnostic report must be inspected. The results obtained with SFL are discussed in [2]. Following the generated ranking would lead to the fault after inspecting 20 statements on average, resulting in a median (and average) SCORE of 14%.

DEPUTO combines SFL with MBSD to refine the ranking of implicated components. To understand how well MBSD filters statements from the ranking, we first study the number of implicated components. Figure 3 contrasts the components implicated by either approach with those blamed by both. It can be seen that MBSD significantly reduces the number of components when compared with SFL. Furthermore, neither approach subsumes the other. Restricting the debugging process to those statements that are implicated by both approaches, the average number of statements reduces from 36 (20 if considering only until the fault is hit) to 8. Hence, the total number of relevant statements reduces considerably. Note, however, that Figure 3 does not imply that SFL's contribution is negligible; although it implicates more components, it also builds a ranking that may quickly lead to the fault.

Since the diagnostic report obtained from DEPUTO is a ranked list of likely faulty components, its size alone is not a good indicator for its quality. Instead, we employ the SCORE metric as defined in the previous section to evaluate our results. Figure 4 visualizes the percentage of located bugs for different fractions of inspected code. Our approach outperforms the individual approaches as well as the simple statistics-based fault localization technique proposed in [14], where different combinations of union and intersection of "similar" passing and failing test runs are computed. This can be attributed to the improved ranking mechanisms built into our algorithm that is more robust with respect to overlapping passing and failing executions. Our combined approach also improves on SOBER [9] and CROSSTAB [16], which are statistical approaches based on hypothesis testing that have been shown to dominate other recent bug detectors. For instance, if up to 10% of the program would have been inspected, DEPUTO would locate 71% of the faults, whereas SOBER and CROSSTAB would yield only 51% and 53%, respectively.

	explain	Δ-slicing	DEPUTO
v1	49	9	7
v11	64	7	7
v31	24	7	7
v40	25	–	16
v41	32	12	6

Table 1: Individual SCORES

Δ-slicing and explain [7] are two techniques for fault localization that exploit differences between passing and failing abstract program executions traces found by a model checker. Table 1 compares our results to the published individual results for all five versions of *TCAS* reported in [7]. We conclude that DEPUTO is far superior to explain (which requires to explore 24–64% of a program) and performs competitive with respect to Delta slicing (within 5%), yet at reduced complexity. However, to understand whether our approach is indeed much better than the ones presented in [7] more experimentation is needed.

Our combined framework also reduces the time required by MBSD from $185s$ [11] to $73.2s$ on average, representing an average speed-up of 2.5 times. The time required by the SFL part of our algorithm is negligible.

6. RELATED WORK

The most similar work to the one presented in this paper is described in [10] where a statistic-based approach is used to rank the set of candidates given by MBSD. While the average computational complexity in [10] is essentially the same as when the model-based approach is used alone, DEPUTO requires significantly less time.

In model-based reasoning, the program model is typically generated from the source code, as opposed to the traditional application of model based diagnosis where the model is obtained from a formal specification of the (physical) system [13]. In [11] an overview of different models for MBSD is given, concluding that the model generated by means of abstract interpretation leads to good results while not suffering from the computational complexity inherent to more precise analysis techniques [11]. Recently, model-based techniques have also been proposed to isolate specific faults

stemming from incorrect implementation of high-level conceptual models [18], where mutations are applied to state machine models to detect conceptual errors, such as incorrect control flow and missing or additional features found in the implementation. Other approaches that fit into this category include `explain` [7] and Δ-slicing [7], which are based on comparing execution traces of correct and failed runs using model checkers.

Combining program execution and symbolic evaluation has been proposed to infer possible errors [6]. Similar to MBSD, a symbolic, under-constrained representation of a program execution and memory structures are built. Instead of using fault probabilities to guide diagnosis, only those candidate explanations that definitively imply a test failure are flagged.

Apart from the model-based approaches, many techniques based on statistical analysis of dynamic program behavior exist. Tarantula [8] obtains program spectra from test executions and graphically visualizes a fault-proneness indicator based on participation of individual statements in passing and failing runs. SOBER [9] is a statistical debugging tool which analyzes traces of predicate evaluations and produces a ranking by contrasting the evaluation bias of each predicate in failing cases against those in passing cases. CROSSTAB [16] exploits the joint distribution of two variables derived from coverage information of different program executions to compute a ranked list of possible faults. In [3] a dynamic modeling approach to fault localization based on logic reasoning over program traces is presented. In [17], discrepancies between execution time spectra obtained from correct and failing tests are used to locate possible faults. In contrast to the algorithm proposed in this paper, the aforementioned techniques do not exploit information about the anticipated behavior of a program and hence rely on external tools to assess the outcome of individual executions, such as automatic error detection based on program invariants [1].

7. CONCLUSIONS & FUTURE WORK

We have shown that the accuracy of spectrum-based fault localization increases significantly when combined with approaches that make use of a model to yield valid explanations for observed failures. Our unique combination of semantics-based analysis as undertaken in model-based software debugging and dynamic aspects obtained from program execution spectra has proved to greatly focus debugging efforts to relevant parts of a program. Overall, a reduction of suspect program fragments to less than 8% of the complete program has been achieved on our test suite, outperforming both individual techniques and most other state of the art techniques. Furthermore, an average speed-up of 2.5 compared to the model-based approach has been observed. We have further shown that our approach is among the state of the art automated debugging tools.

Several issues for further research remain. On the dynamic analysis side, introducing machine learning techniques to infer likely invariants that can then be used to reduce the number of implicated components by the spectrum-based fault localization approach are possible avenues worth further exploration. On the MBSD side, connecting the lower-level models that reflect most details of a program to high-level conceptual models to detect a more diverse set of faults seems promising to broaden the scope of applicability. Due to limitations of our implementation of the model-based approach, we were not yet able to apply the combined framework to a comprehensive test suite. (Our system currently has only limited support for interfaces to external systems, such as libraries, I/O, and multi-threading.) We plan to apply it to more realistic programs to gain further insights and to identify possible further improvements of our debugging algorithms.

8. REFERENCES

[1] R. Abreu, A. González, P. Zoeteweij, and A. J. C. van Gemund. Automatic software fault localization using generic program invariants. In *Proc. SAC'08 - SE Track*. ACM Press, 2008.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *TAIC PART'07*. IEEE CS, 2007.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An observation-based model for fault localization. In *Proc. WODA'08*. ACM Press, 2008.

[4] C. Böttcher. No faults in structure? How to diagnose hidden interaction. In *Proc. IJCAI*, 1995.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL'77*. ACM Press, 1977.

[6] D. R. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proc. ISSTA'07*. ACM Press, 2007.

[7] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.

[8] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE'05*. ACM Press, 2005.

[9] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *FSE'05*. ACM Press, 2005.

[10] W. Mayer, R. Abreu, M. Stumptner, and A. J. C. van Gemund. Prioritizing model-based debugging diagnostic reports. In *Proc. DX'08*, 2008.

[11] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *ASE'08*. ACM Press, 2008.

[12] M. Musuvathi and D. R. Engler. Some lessons from using static analysis and software model checking for bug finding. *ENTCS*, 89(3), 2003.

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

[14] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE'03*. ACM Press, 2003.

[15] RTI. Planning report 02-3: The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, NIST, 2002.

[16] E. Wong, T. Wei, Y. Qi, and L. Zhao. A crosstab-based statistical method for effective fault localization. In *Proc. ICST'08*. IEEE CS, 2008.

[17] C. Yilmaz, A. Paradkar, and C. Williams. Time will tell: Fault localization using time spectra. In *Proc. ICSE '08*. ACM Press, 2008.

[18] C. Yilmaz and C. Williams. An automated model-based debugging approach. In *ASE'07*. ACM Press, 2007.

[19] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. C. van Gemund. Diagnosis of embedded software using program spectra. In *Proc. ECBS'07*. IEEE CS, 2007.