A Dynamic Modeling Approach to Software Multiple-Fault Localization[*]

Rui Abreu and **Peter Zoeteweij** and **Arjan J.C. van Gemund**

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

Abstract

Current model-based approaches to software debugging use static program analysis to derive a model of the program. In contrast, in the software engineering domain diagnosis approaches are based on analyzing dynamic execution behavior. We present a model-based approach where the program model is derived from dynamic execution behavior, and evaluate its diagnostic performance for both synthetic programs and the Siemens software benchmark, extended by us to accommodate multiple faults. We show that our approach outperforms other model-based software debugging techniques, which is partly due to the use of De Kleer's intermittency model to account for the variability of software component behavior.

1 Introduction

Automatic software fault localization techniques aid developers to pinpoint the root cause of failures, thereby reducing the debugging effort. Two major approaches can be distinguished, (1) the spectrum-based fault localization (SFL) approach, a statistical approach that correlates dynamic software component activity (i.e., execution traces) with program failures [1; 12; 13], and (2) the model-based diagnosis or debugging (MBD) approach, which deduces component failure through logic reasoning over a static model of the program [4; 5; 6; 7; 8; 9; 15; 16; 20].

Because of its low computational complexity and absence of modeling requirements, SFL has gained large popularity in the software engineering community. Although inherently not restricted to single faults, in most cases these statistical techniques are applied and evaluated in a single-fault context, such as the Siemens benchmark set [10], which is seeded with only 1 fault per program (version). In practice, however, the defect density of even small programs typically amounts to multiple faults. Although the root cause of a particular program failure need not constitute multiple faults that are acting *simultaneously*, many failures will be caused by *different* faults. Hence, the problem of multiple-fault localization (diagnosis) deserves detailed study.

Unlike SFL, MBD inherently considers multiple faults. However, the logic models of software systems that are used in the diagnostic inference are typically based on static program analysis. Consequently, they do not consider dynamic execution behavior, such as (data-dependent) conditional control flow, which, in contrast, forms the essence of the SFL approach. Aimed to combine the best of both worlds, in this paper we present an approach that exploits the dynamic, execution trace-based observation approach from SFL, to derive models and observations as input to MBD to produce multiple-fault diagnoses.

To illustrate the potential advantages of a multiple-fault approach, consider a triple-fault program with faulty components c_1, c_2, and c_3. Whereas (*ideally*) a single-fault approach such as SFL would produce multiple single-fault diagnoses like $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \ldots\}$ (component indices, ordered in terms of statistical similarity), a multiple-fault approach would simply produce one single multiple-fault diagnosis $\{\{1, 2, 3\}\}$. This single diagnosis unambiguously reveals the actual triple fault, which, additionally, measures the potential for debugging parallelism [11], whereas in the former case it is not obvious how many faults are actually present.

This paper makes the following contributions:

- We present our multiple-fault diagnosis method which combines a dynamic modeling and observation approach known from SFL with a diagnostic reasoning approach from MBD.
- We evaluate our approach using the Siemens set benchmark, extended by us to accommodate multiple faults.
- We evaluate the merit of two specific strategies for updating the probabilities of diagnosis candidates, based on De Kleer's intermittent fault model [4], to account for the fact that faulty (software) components very often exhibit nominal behavior.
- We compare our approach to related reasoning approaches (AIM [15], Δ-slicing [8], and `explain` [8]) for the Siemens set program `tcas` (their common benchmark).

Our experiments show that strategies that exploit (intermittency) information to exonerate components involved in

[*]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.

passed runs outperform those that do not include such information. Furthermore, experiments using the `tcas` program show that our approach allows that more bugs can be solved if limited debugging time is available.

2 Preliminaries

In this section we introduce the concepts and definitions used throughout this paper.

2.1 Basic Definitions

Definition 1 *A diagnostic system DS is defined as the triple $DS = \langle SD, COMPS, OBS\rangle$, where SD is a propositional theory describing the behavior of the system, $COMPS = \{c_1, \ldots, c_M\}$ is a set of components in SD, and OBS is a set of observable variables in SD.*

With each component $c_m \in COMPS$ we associate a *health variable* h_m which denotes component health. The health states of a component are healthy (*true*) and faulty (*false*).

Definition 2 *An h-literal is h_m or $\neg h_m$ for $c_m \in COMPS$.*

Definition 3 *An h-clause is a disjunction of h-literals containing no complementary pair of h-literals.*

Definition 4 *A conflict of $(SD, COMPS, OBS)$ is an h-clause of negative h-literals entailed by $SD \cup OBS$.*

Definition 5 *Let S_N and S_P be two disjoint sets of components indices, faulty and healthy, respectively, such that $COMPS = \{c_m \mid m \in S_N \cup S_P\}$ and $S_N \cap S_P = \emptyset$. We define $d(S_N, S_P)$ to be the conjunction:*

$$(\bigwedge_{m\in S_N} \neg h_m) \wedge (\bigwedge_{m\in S_P} h_m)$$

A diagnosis candidate is a sentence describing one possible state of the system, where this state is an assignment of the status healthy or not healthy to each system component.

Definition 6 *A diagnosis candidate for $(SD, COMPS, OBS)$, given an observation term obs over variables in OBS, is $d(S_N, S_P)$ such that*

$$SD \wedge obs \wedge d(S_N, S_P) \not\models \perp$$

In the remainder we refer to $d(S_N, S_P)$ simply as d, which we identify with the set S_N of indices of the negative literals.

Definition 7 *A diagnosis $D = \{d_1, \ldots, d_k, \ldots, d_K\}$ is an ordered set of all K diagnosis candidates, for which $SD \wedge obs \wedge d_k \not\models \perp$*

2.2 Model-based Diagnosis

In this section we describe the principles underlying model-based software diagnosis as far as relevant to this paper.

Consider the simple program function in Figure 1, which is supposed to be composed of three inverting statements (with a fault in statement 3), resembling a circuit with three logical inverters[1] . The function takes one input x, and returns two outputs y_1, y_2. A *weak* model of each inverter statement, which only specifies nominal (required) behavior, is given by the proposition

```
     (y1,y2) 3inv(bool x) {
1.     w  = !x
2.     y1 = !w;
3.     y2 = w; //fault: ! missing
       return (y1,y2); }
```

Figure 1: A defective function

$$h \Rightarrow y = \neg x$$

Given the data dependencies of the program, the interconnection topology of the three inverting components is easily obtained, yielding the (combined) program model

$$h_1 \Rightarrow w = \neg x$$
$$h_2 \Rightarrow y_1 = \neg w$$
$$h_3 \Rightarrow y_2 = \neg w$$

Computing Diagnoses

Consider the observation $obs = ((x, y_1, y_2) = (1, 1, 0))$. From the model, it follows

$$h_1 \Rightarrow \neg w$$
$$h_2 \Rightarrow \neg w$$
$$h_3 \Rightarrow w$$

which equals

$$(\neg h_1 \vee \neg w) \wedge (\neg h_2 \vee \neg w) \wedge (\neg h_3 \vee w)$$

Resolution yields the following conjunction of conflicts

$$(\neg h_1 \vee \neg h_3) \wedge (\neg h_2 \vee \neg h_3)$$

meaning that (1) at least c_1 or c_3 is at fault, and (2) at least c_2 or c_3 is at fault. The minimal diagnoses are given by the minimal hitting set [18], yielding

$$\neg h_3 \vee (\neg h_1 \wedge \neg h_2)$$

Thus either c_3 is at fault (single fault), or c_1 and c_2 are at fault (double fault), as well as a number of other double-faults ($\neg h_2 \vee h_3$, $\neg h_1 \vee h_3$), and a triple fault ($\neg h_1 \vee h_1 \vee h_3$), which, however, are *subsumed* by the previous two *minimal* diagnoses. Consequently, $D = \{\{3\}, \{1, 2\}\}$.

Ranking Diagnoses

The fact that models do not always specify all possible behavior (e.g., weak models), and that usually only limited observations are available typically leads to diagnoses with many solutions. However, not all solutions are equally probable, allowing them to be ranked in order of probability of being the actual fault state.

Let $\Pr(\{j\})$ denote the *a priori* probability that a component c_j is at fault. Although this value is typically component-specific, in the above inverter example we assume $\Pr(\{j\}) = p$ (where we arbitrarily set $p = 0.01$). Assuming components fail independently, and in absence of any observation,

[1]Note that in this approach we assume the presence of a correct model of the components (in contrast to e.g. [14])

the prior probability a particular diagnosis d_k is correct is given by

$$\Pr(d_k) = \prod_{j \in S_N} \Pr(\{j\}) \cdot \prod_{j \in S_P} (1 - \Pr(\{j\}))$$

In order to compute the posterior probability given an observation we use Bayes' rule, which can be applied iteratively for multiple observations

$$\Pr(d_k|obs) = \frac{\Pr(obs|d_k)}{\Pr(obs)} \cdot \Pr(d_k)$$

The denominator $\Pr(obs)$ is a normalizing term that is identical for all d_k and thus needs not be computed directly. $\Pr(obs|d_k)$ is defined as

$$\Pr(obs|d_k) = \begin{cases} 0 & \text{if } d_k \text{ and } obs \text{ are inconsistent} \\ 1 & \text{if } d_k \text{ implies } obs \\ \varepsilon & \text{if neither holds} \end{cases}$$

Many policies exist for ε [3]. For the purpose of the above example, we adopt the classical policy $\varepsilon = 1/dx$ where dx is the number of observations that can be explained by diagnosis d_k. As there are 4 possible observations that can be explained by $\{3\}$, and 8 that can be explained by $\{1, 2\}$, it follows

$$\Pr(obs|\{3\}) = \tfrac{1}{4}\,;\, \Pr(obs|\{1,2\}) = \tfrac{1}{8}$$

Hence, the diagnosis is given by (after normalization)

d_k	$\Pr(d_k)$
$\{3\}$	0.995
$\{1,2\}$	0.005

3 Observation-based Modeling

The above approach is dependent on the existence of a model of the program, which would have to be derived from the system specifications. Even if a model were available for each component (statement), only for the simplest of programs (such as our example program) a program model could be extracted based on static dependence analysis. In this section we present our dynamic, observation-based diagnosis approach. This model is used to compute the set of valid diagnoses, which will then be ranked according to the likelihood that they explain the failures.

3.1 Observations

Observations are collected as abstractions of execution traces, in SFL called program spectra. A program spectrum is a collection of observations that provides a specific view on the dynamic behavior of software. This data is collected at run-time, and typically consists of a number of counters or flags for the different components of a program. In the context of this paper we use the so-called hit spectra, which indicate whether a component (statement) was involved in a run.

Definition 8 *Let M be the number of components, and N the number of execution runs. Let O denote the* $N \times (M+1)$ observation matrix. *For $j \leq M$, the element o_{ij} is equal to* 1 (true) *if component j was observed to be involved in the execution of run i, and* 0 (false) *otherwise. The element $o_{i,M+1}$ is equal to* 1 (true) *if run i failed, and* 0 (false) *otherwise. The rightmost column of O is also denoted as e (the error vector).*

Note that at least one faulty component has to be involved in the computation of a failed run. From O it is also possible to derive the probability r that a component is actually executed in a run (expressing code coverage), and the probability g that a faulty component is actually exhibiting good behavior (expressing fault coverage, also known as the "goodness" parameter g [4]).

3.2 Computing Diagnoses

Unlike the MBD approach mentioned earlier, which statically deduces information from the program source, O is the *only*, dynamic source of information, from which *both* a model, and the input-output observations are derived. Apart from exploiting dynamic information, this approach only requires a generic component model, avoiding the need for detailed functional modeling or relying, e.g., on invariants or pragmas for model information. Note, however, that this default model can easily be extended when more detailed information is available.

Abstracting from particular component behavior, each component c_j is modeled by the weak model

$$h_j \Rightarrow (x_j \Rightarrow y_j)$$

where h_j models the health state of c_j and x_j, y_j model its input and output variable value *correctness* (i.e., we abstract from actual variable *values*, in contrast to the earlier example). This weak model implies that a healthy component c_j translates a correct input x_j to a correct output y_j. However, a faulty component or input *may* lead to an erroneous output.

As each row in O specifies which components were involved, we interpret a row as a "run-time" model of the program as far as it was considered in that particular run. Consequently, O is interpreted as a sequence of typically different models of the program, each with its particular observation of input/output correctness. The overall diagnosis can be viewed as a sequential diagnosis approach that incrementally takes into account new structural program (and pass/fail) evidence with increasing N. A single row $O_{n,*}$ corresponds to the (sub)model

$$\begin{aligned} h_m &\Rightarrow (x_m \Rightarrow y_m), \text{ for } m \in T_n \\ x_{t_i} &= y_{t_{i-1}}, \text{ for } i \geq 2 \\ x_{t_1} &= \text{true} \\ y_{t'} &= \neg e_n \end{aligned}$$

where $T_n = \{m \in \{1, \ldots, M\} \mid o_{nm} = 1\}$ denotes the well-ordered set of component indices involved in computation n, t_i denotes the i^{th} element in this ordering, (i.e., for $i \leq j, t_i \leq t_j$), t' denotes its last element. The resulting component chain logically reduces to

$$\bigwedge_{m \in T_n} h_m \Rightarrow \neg e_n$$

For example, consider the row ($M = 5$)

c_1	c_2	c_3	c_4	c_5	e
1	0	0	1	0	1

This corresponds to a model where components c_1, c_4 are involved. As the order of the component invocation is not given (and with respect to our above weak component model is irrelevant), we derive the model

$$h_1 \Rightarrow (x_1 \Rightarrow y_1)$$
$$h_4 \Rightarrow (x_4 \Rightarrow y_4)$$
$$x_4 = y_1$$
$$x_1 = \text{true}$$
$$y_4 = \neg e_n$$

In this chain the first component c_1 is assumed to have correct input ($x_1 = \text{true}$, typical of a proper test), its output feeds to the input of the next component c_4 ($x_4 = y_1$), whose output is measured in terms of e_n ($y_4 = \neg e_n$). This chain logically reduces to

$$h_1 \wedge h_4 \Rightarrow \text{false}$$

If this were a passing computation ($h_1 \wedge h_4 \Rightarrow \text{true}$) we could not infer anything (apart from the exoneration when it comes to probabilistically rank the diagnosis candidates as explained in next section). However, as this run failed this yields

$$\neg h_1 \vee \neg h_4$$

which, in fact, is a conflict. In summary, each failing run in O generates a conflict according to

$$\bigvee_{m \in T_n} \neg h_m$$

As in the former MBD approach, the conflicts are then subject to a hitting set algorithm that generates the diagnostic candidates.

To illustrate this concept, again consider the example program. For the purpose of the spectral approach we assume the program to be run two times where the first time we consider the correctness of y_1 and the second time y_2. This yields the observation matrix O below

c_1	c_2	c_3	e	
1	1	0	0	obs_1
1	0	1	1	obs_2

From obs_2, it follows

$$\neg h_1 \vee \neg h_3$$

which equals the first conflict from the earlier MBD approach, and the diagnosis trivially comprises the two single faults $\{1\}$ ($\neg h_1$) and $\{3\}$ ($\neg h_3$). Compared to the earlier MBD approach, the second conflict ($\neg h_2 \vee \neg h_3$) is missing due to the fact that no additional knowledge is available on component behavior and component interconnection. Although this would suggest that the dynamic approach yields lower diagnostic performance than the earlier MBD approach, note that the example program is ideally suited to static analysis, whereas real programs feature extensive control flow, rendering the static approach extremely difficult. However, if, for some reason, we were able to capture the second conflict in terms of an execution trace according to

c_1	c_2	c_3	e	
0	1	1	1	obs_3

then our observation-based approach would yield exactly the same set of minimal diagnoses.

Note that, e.g., unlike dynamic slicing [21] and constraint-based models [17], we do not exploit actual data dependencies between components but execution patterns.

3.3 Ranking Diagnoses

Similar to the incremental compilation of conflicts per run we compute the posterior probability for each candidate based on the pass/fail observation *obs* for each sequential run using Bayes' rule as described in Section 2.2. In the following we will distinguish between three ε policies. The first policy, denoted $\varepsilon^{(0)}$ is similar to the classical MBD policy, and is defined as follows

$$\varepsilon^{(0)} = \begin{cases} \frac{E_P}{E_P + E_F} & \text{if run passed} \\ \frac{E_F}{E_P + E_F} & \text{if run failed} \end{cases} \qquad (1)$$

where $E_P = 2^M$ and $E_F = (2^l - 1) \cdot 2^{M-l}$ are the number of passed and failed observations that can be explained by diagnosis d_k, respectively, and $l = |d_k|$ is the number of faulty components in the diagnosis. Note that this policy is slightly different from the one in Section 2.2, as the lack of component interconnection information allows more diagnoses (component combinations) as likely explanations for pass/fail outcomes.

A disadvantage of this classical policy is that passed runs, apart from making single faults more probable than multiple faults, do not help much in pinpointing the fault location. This has to do with the fact that all diagnoses are possible when a run passes due to the weak fault model (the 2^M term in Eq. 1). In addition, there is no way to distinguish between diagnoses with the same cardinality, because the terms are merely a function of the cardinality of the diagnosis candidate.

An approach to account for the fact that, similar to statistical approaches for fault localization, components involved in passed computations should to some extent be exonerated, is by extending the component model with an intermittent failure model, as introduced by De Kleer [4]. As in software components it is quite usual that a faulty component exhibits correct behavior, we include statistical information on the probability that a faulty component c exhibits correct behavior. Let $g(d_k)$ denote the aforementioned ("goodness") probability that faulty components in d_k are exhibiting good behavior. In the following we distinguish between two different policies, which we refer to as $\varepsilon^{(1)}$, and $\varepsilon^{(2)}$, which are defined as follows

$$\varepsilon^{(1)} = \begin{cases} g(d_k) & \text{if run passed} \\ 1 - g(d_k) & \text{if run failed} \end{cases}$$

and

$$\varepsilon^{(2)} = \begin{cases} g(d_k)^t & \text{if run passed} \\ 1 - g(d_k)^t & \text{if run failed} \end{cases}$$

where t is the number of faulty components according to d_k involved in the run i

$$t = \prod_{j \in d_k} [o_{ij} = 1]$$

We propose policy $\varepsilon^{(2)}$ as a variant of $\varepsilon^{(1)}$, which is due to De Kleer [4]. It approximates the probability $\sum_{j \in d_k} g_j$ that the components in d_k all exhibit good behavior by $g(d_k)^t$, assuming that all components of d_k have equal goodness probabilities. In both strategies we use

$$g(d_k) = \frac{\sum_{i=1..N} [(\bigvee_{j \in d_k} o_{ij} = 1) \wedge e_i = 0]}{\sum_{i=1..N} [\bigvee_{j \in d_k} o_{ij} = 1]}$$

where $[\cdot]$ is Iverson's operator ($[\text{true}] = 1$, $[\text{false}] = 0$). In the above two policies $g(\cdot)$ is assumed to be different from 0. Otherwise a strong exoneration factor is included, as a passed run would set $\Pr(d_k) = 0$, for all diagnosis candidates. Furthermore, one can think of additional policies, such as combining either $\varepsilon^{(1)}$ or $\varepsilon^{(2)}$ with $\varepsilon^{(0)}$. Note, however, that in this case the variable term in $\varepsilon^{(0)}$ is a function of the diagnosis' cardinality only. As the prior probabilities p already differentiate diagnoses with different cardinalities, such a combined policy would not produce a different ranking than those obtained by either $\varepsilon^{(1)}$ or $\varepsilon^{(2)}$.

	$\varepsilon^{(0)}$	$\varepsilon^{(1)}$	$\varepsilon^{(2)}$
$\Pr(\{1\})$	0.5	0.2	0.2
$\Pr(\{3\})$	0.5	0.8	0.8

(a) After obs_1 and obs_2

	$\varepsilon^{(0)}$	$\varepsilon^{(1)}$	$\varepsilon^{(2)}$
$\Pr(\{3\})$	0.98	0.99	0.999
$\Pr(\{1,2\})$	0.02	0.01	0.001

(b) After obs_1, obs_2, and obs_3

Figure 2: Probabilities updates

Returning to the example of Section 2.2, Figure 2 lists the probabilities resulting from the various ε policies for the diagnoses obtained after obs_1 and obs_2 only (Figure 2(a)) and after obs_3 (Figure 2(b)). In the first case, the classical policy cannot distinguish between c_1 and c_3 while the g policies exploit the additional information provided by the exonerating observation obs_1. When obs_3 is included c_1 is no longer a valid diagnosis by itself, and is eliminated from the (hitting) set of valid diagnosis candidates. Hence, all policies favor c_3 as most likely candidate, due to (1) the lower prior probability of the double fault ($\varepsilon^{(0,1,2)}$) and (2) the exoneration by passed runs ($\varepsilon^{(1,2)}$).

4 Theoretical Evaluation

In order to gain understanding of the effects of the various parameters on the diagnostic performance of the different policies, in this section we use a simple, probabilistic model of program behavior that is directly based on N, M, r, and g. Without loss of generality we model the first C of the M components to be at fault (C for fault cardinality). For each run each component has probability $r = 0.6$ (which go in accordance with the value measured for our software benchmark set of faults, see next section) to be involved in that run. If a selected component is faulty, the probability of exhibiting nominal ("good") behavior equals g. When either of the C components fails, the run will fail. We study the performance of the ε policies defined previously for observation matrices that are randomly generated according to the above model.

4.1 Performance Metrics

Before evaluating the results, we first present our performance metric. Fault localization techniques aim at helping developers in finding bugs quickly, and a metric to evaluate such techniques is to measure the amount of code a developer would have to inspect before (but not including) finding the fault cause, *wasted effort* W. It is defined as the number of inspected components divided by the total number of components (M). In our computation of W we assume that after each inspection, the test set is rerun, possibly leading to a new ranking (without the most recently removed fault). For example, suppose a triple-fault program ($M = 6$, and c_1, c_2, and c_3 faulty) for which the following diagnosis $D = \{\{1, 2, 6\}, \{3, 4, 5\}\}$ is obtained. This diagnosis induces a wasted effort of $W = 33\%$ as c_6 in the first candidate is inspected in vain, as well as, on average two out of three inspections in the second candidate (in this example we assumed that rerunning the test set did not change the second candidate $\{3, 4, 5\}$). For example, had the two components in the second diagnosis candidate been inspected, then $W = 50\%$.

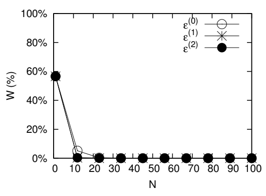
In contrast to related work, we measure W instead of effort [1; 19] so that the performance metric's scale is independent of the number of faults in the program. Another reason not to adopt the aforementioned score metric is that in our synthetic model we do not have program dependence graph information. For the example given above, the effort as defined in [1; 19] would be 100%.

4.2 Diagnosis Optimality

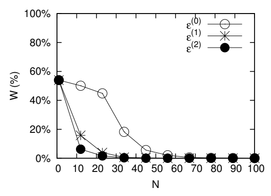
As mentioned in the Introduction, under ideal circumstances our multiple-fault approach produces one single multiple-fault diagnosis $\{1, \ldots, C\}$. This optimal result (shown in [2]) is obtained (for programs where each component has an independent, non-zero probability of being involved in any run) when $N \to \infty$. This can be seen through the following argument. Consider a C-fault program. While for small N the minimal hitting set will still contain many members (components) other than the C faulty components, by increasing N the probability that a non-faulty component will still be included steadily decreases. Let f denote the probability of a run failing (derived as function of r, g, C in [2]). For the hitting set analysis only failing runs matter. For those $N_F = f \cdot N$ failing runs the C-fault candidate is by definition within the set of candidates that "survive" those runs (whose chain is still unbroken). However, the probability that other components can be involved in a candidate is less than unity, which forms the basis of those candidates' eventual elimination. In the following experiments this dependency of W on N is further studied.

4.3 Experimental Results

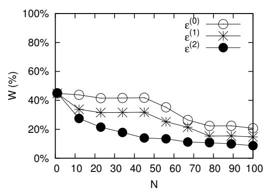
In this section we experimentally study the diagnostic performance of the different policies. For that purpose, a synthetic observation matrix generator is implemented, which takes into account N, g, and C. Although we verified the influence of these parameters, in the following M is fixed to 20 and r to 0.6 as they do not change our conclusions.



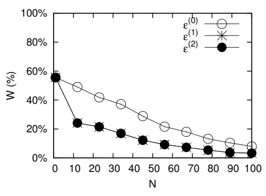
(a) $C = 1$ and $g = 0.1$



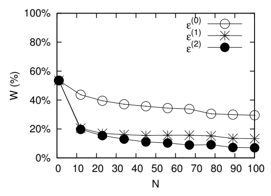
(b) $C = 2$ and $g = 0.1$



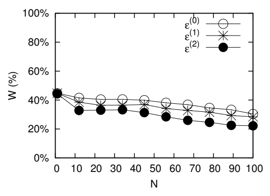
(c) $C = 5$ and $g = 0.1$



(d) $C = 1$ and $g = 0.9$



(e) $C = 2$ and $g = 0.9$



(f) $C = 5$ and $g = 0.9$

Figure 3: ε-policies diagnostic performance

Figure 3 contains plots of W versus N for $C = 1$, $C = 2$ and $C = 5$. Each measurement represents an average over 1,000 sample matrices. The plots show that W for $N = 1$ is similar to r, which follows from the fact that there are on average $(M - C) \cdot r$ components which would have to be inspected in vain. For sufficiently large N all policies produce an optimal diagnosis, see previous section. For small number of runs N, given the fact that it does not distinguish between diagnosis with the same fault cardinality (see Section 3.3), $\varepsilon^{(0)}$ is the worst performing policy. For $C \geq 2$, $\varepsilon^{(2)}$ outperforms $\varepsilon^{(1)}$, suggesting that information on the number of components involved a run should be included (t in $\varepsilon^{(2)}$). For $C = 1$, as $t = 1$ the performance of $\varepsilon^{(1)}$ equals the one of $\varepsilon^{(2)}$.

Furthermore, from the plots we verify that the higher C the more runs N are needed to attain optimal diagnostic performance. As an example, for $g = 0.1$, $r = 0.4$, and $C = 1$, 13 runs would be enough to yield a perfect diagnosis, whereas for $C = 5$, 250 runs would be needed. We have determined the value of N (N_F) for which our C-cardinality fault remains as the only candidate, i.e., a perfect multiple-fault diagnosis $\{1, \ldots, C\}$. Table 1 shows the values of N (N_F) where optimality is reached for different values of C and g. Apart from a scaling due to g one can clearly see the exponential impact of C on N_F and N (shown in [2]).

g	0.1					0.9				
C	1	2	3	4	5	1	2	3	4	5
N	13	31	90	120	250	200	300	500	1000	1700
N_F	5	19	71	111	245	12	36	84	219	459

Table 1: Optimal N^* for perfect diagnosis ($r = 0.6$)

5 Experimental Evaluation

In this section we assess the diagnostic capabilities of the dynamic modeling approach for real programs. For this purpose, we use the well-known Siemens set [10], which contains 132 faulty versions of 7 C programs with extensive test suites. Table 2 summarizes the characteristics of the Siemens set, where M corresponds to the number of lines of code (components in this context).

Program	Faulty Versions	M	N	Description
`print_tokens`	7	539	4,130	Lexical Analyzer
`print_tokens2`	10	489	4,115	Lexical Analyzer
`replace`	32	507	5,542	Pattern Recognition
`schedule`	9	397	2,650	Priority Scheduler
`schedule2`	10	299	2,710	Priority Scheduler
`tcas`	41	174	1,608	Altitude Separation
`tot_info`	23	398	1,052	Information Measure

Table 2: The Siemens benchmark set

For our experiments, we have extended the Siemens set with program versions in which we can activate arbitrary combinations of faults. For this purpose, we limit ourselves to a selection of 102 out of the 132 faults, based on criteria such as faults being attributable to a single line of code, to enable unambiguous evaluation. The observation matrices are obtained using the GNU `gcov`[2] profiling tool.

Using this extended Siemens set, we evaluate our dynamic modeling approach in two ways: first, in Section 5.1, we measure its diagnostic performance on single and multiple-fault programs for the three ε strategies outlined in Section 2.2. Next, in Section 5.2 we compare this performance against other diagnosis techniques. Here we use single-fault versions of the `tcas` program, which is the common program used in literature to evaluate these other techniques.

5.1 Results

Table 3 lists the wasted effort W, as defined in Section 4.1, incurred by the dynamic modeling approach and strategies $\varepsilon^{(0)}$, $\varepsilon^{(1)}$, and $\varepsilon^{(2)}$ for debugging single, double, and multiple-fault programs. Like in Section 4.3, we aimed at $C = 5$ for the multiple fault-cases, but for `print_tokens` insufficient faults are available, and for `print_tokens2` and `replace` our current implementation of the hitting set algorithm practically prevents analyzing combinations of more

[2]http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

	print_tokens			print_tokens2			replace			schedule			schedule2			tcas			tot_info		
C	1	2	4	1	2	4	1	2	3	1	2	5	1	2	5	1	2	5	1	2	5
versions	4	6	1	10	43	100	23	100	100	7	20	11	9	43	100	30	100	100	19	100	100
$\varepsilon^{(0)}$	13.54	17.84	22.63	21.39	25.72	29.38	15.78	24.64	28.06	16.48	22.49	27.09	30.17	28.15	29.58	27.39	26.49	27.74	14.18	17.48	26.45
$\varepsilon^{(1)}$	1.25	2.54	5.19	3.31	5.94	10.51	2.99	5.29	7.14	0.83	1.62	3.07	24.86	32.50	38.58	16.48	23.39	29.52	5.05	8.94	17.20
$\varepsilon^{(2)}$	1.25	2.50	5.01	3.62	6.67	12.08	2.90	5.31	7.28	0.83	1.95	5.13	23.00	29.92	36.21	16.48	23.32	29.49	6.10	11.20	19.94

Table 3: Wasted effort W [%] for $\varepsilon^{(0)}$, $\varepsilon^{(1)}$, and $\varepsilon^{(2)}$ on combinations of 1-5 faults from the Siemens set

than four and three faults, respectively[3]. The hitting set computation is aborted after all diagnosis candidates with cardinality C' have been generated. To simulate a more or less realistic debugging scenario, where the actual number of faults is unknown, we set $C' = \max(C, 3)$. All measurements except for the four-fault version of `print_tokens` are averages over 100 versions, or over the maximum number of combination available, where we verified that all faults are active in at least one failed run.

Similar to what we observed in Section 4.3, in most cases $\varepsilon^{(1,2)}$ lead to significantly better diagnoses than $\varepsilon^{(0)}$ because they exonerate components that are involved in passed runs. The (minor) differences between the results for $\varepsilon^{(1)}$ and $\varepsilon^{(2)}$ at $C = 1$ are due to the different ways in which these strategies handle the diagnosis candidates of cardinality 2 and 3 that result from setting $C' = 3$.

Contrary to the results in Section 4.3, the improvement of $\varepsilon^{(2)}$ over $\varepsilon^{(1)}$ is marginal at best. We expect that this can be explained by using $g(d_k)^t$ to approximate the product of the goodness parameters of the individual components in a diagnosis d_k, as explained in Section 3.3. In the context of strategy $\varepsilon^{(2)}$, this entails using different goodness parameters for the same component as it occurs in different diagnoses, converging to the fraction of all runs that have passed as the diagnosis cardinality increases. The influence of these effects, and the unexpected superior performance of $\varepsilon^{(0)}$ in the particular cases of `schedule2` for $C = 2$ and $C = 5$, and `tcas` for $C = 5$ require further investigation.

5.2 Comparison

In the following we compare the diagnostic performance of our approach with AIM, nearest neighbor (NN), `explain`, and Δ-slicing techniques (see Section 6 for a discussion). For compatibility with results reported for those techniques, we will use the effort, or *score* metric [1; 19] instead of wasted effort W which amounts to the percentage of lines of code that need *not* be examined when the diagnosis results are used to guide the search for the fault. Note that the results reported for NN do not involve a full ranking of all statements, but are based on the distance between the fault and the diagnosis in the program dependence graph instead, which is a comparable measure [12].

Our current implementation of the dynamic modeling approach only supports C programs, while the AIM technique has mainly been evaluated for Java programs. The only C program that has been taken into account is `tcas`, which happens to be a common benchmark among the other techniques as well, so for this comparison we limit ourselves to

[3]A novel, statistics-directed improvement is currently under development. Preliminary results indicate orders of magnitude speedup.

Effort	AIM	NN	$\varepsilon^{(0)}$	$\varepsilon^{(1,2)}$
< 1	34	21	0	48
< 10	70	34	0	63
< 20	100	36	0	100
< 30	100	46	0	100
< 40	100	46	0	100
< 50	100	53	0	100
< 60	100	53	0	100
< 70	100	53	0	100
< 80	100	53	97	100
< 90	100	53	100	100
≤ 100	100	100	100	100

Table 4: Cumulative Percentage of Faults found for `tcas`

`tcas`	AIM	`explain`	Δ-slicing	$\varepsilon^{(0)}$	$\varepsilon^{(1,2)}$
v1	0.74	0.51	0.91	0.13	0.99
v11	0.84	0.36	0.93	0.17	0.97
v31	0.77	0.76	0.93	0.17	0.98
v40	0.85	0.75	–	0.17	0.90
v41	0.73	0.68	0.88	0.18	0.99

Table 5: Comparison with distance metrics techniques

that program. Furthermore, the other techniques have only been evaluated for single faults, so we set $C' = C = 1$, and therefore $\varepsilon^{(1)} = \varepsilon^{(2)}$.

Similar to the results in [15], in Table 4 we compare our approach with AIM and NN on `tcas`. As expected, $\varepsilon^{(0)}$ is outperformed by all other techniques. AIM consistently outperforms NN. For an effort of less than 1%, $\varepsilon^{(1,2)}$ outperform AIM, which yields the best results if 10% of the code is inspected. Both techniques find all faults by inspecting less than 20% of the code.

Table 5 compares the different policies used in our approach with AIM, `explain`, and Δ-slicing for 5 versions of `tcas`, because these are the versions to which `explain` and Δ-slicing could be applied to. From the table, we conclude that our approach when using $\varepsilon^{(1,2)}$ consistently outperforms all other techniques, with $\varepsilon^{(0)}$ being the worst performing technique.

6 Related Work

As mentioned in the introduction, automated debugging techniques can be distinguished into statistical and logic reasoning approaches that use program models.

In logic (model-based) reasoning approaches to automatic software debugging, the program model is typically generated from the source code. In [15] an overview of techniques based on automatically generated program models is given. They conclude that the models generated by means of abstract interpretation [14] (AIM approach) are the most accurate for debugging. Basically, a model of the (faulty) program is generated from the source code, e.g., using abstract interpretation, and the test cases specify the expected output. Differences between the program's output and the expected one are used to compute components that when assumed to behave differently explain the observed faulty behavior. Approaches based on model checkers include the `explain` [9], Δ-slicing [8], which are based on comparing execution traces of correct and failed runs. Although model-based diagnosis inherently considers multiple-faults, thus far the above software debugging approaches only consider single faults. Apart from the fact that our approach is multiple-fault, it also differs in the fact that we use program spectra as dynamic information on component activity, which allows us to exploit dynamic execution behavior, unlike static approaches. Furthermore, our approach does not rely on the approximations required by static techniques. In addition, similar to AIM, the approach presented in this paper does not require a formal specification of the program.

Statistical approaches are very attractive from complexity-point of view. Well-known examples are the Tarantula tool [12], the Nearest Neighbor technique [19], the Sober tool [13], and the Ochiai coefficient [1]. Although differing in the way they derive the statistical fault ranking, all techniques are based on measuring program spectra. All these approaches yields single fault explanations, whereas ours also includes multiple faults that explain all failures. This *extra* information may help to debug several failures in parallel, similar to [11] which uses clustering techniques to build sets of runs revealing the same failure.

7 Conclusions and Future Work

In this paper we present a dynamic modeling approach to software fault localization based on abstraction of program traces. The model, along with the set of traces for pass/fail executions is used to reason about observed failures. In contrast to most approaches to software fault diagnosis, which present diagnosis candidates as single explanations, our approach also contains multiple fault explanations in the diagnostic ranking (typical of model-based approaches).

We have evaluated the diagnostic performance of three Bayesian probability update policies, including De Kleer's intermittency model and an extension proposed by us. Empirical results obtained from the widely-used Siemens set of programs, extended by us to accommodate multiple fault programs, show that policies that are able to exonerate components that are involved in passing runs clearly outperform the probability update scheme that is traditionally used in model-based diagnosis.

For future work, we plan to study whether the multiple-fault diagnosis candidates information can be used to efficiently engage several developers to repair the defect(s) in parallel, as well as incorporate our new algorithm to speedup the hitting set computation.

Acknowledgments

We extend our gratitude to Johan de Kleer for discussions which have influenced our multiple-fault reasoning approach.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proc. TAIC PART '07*, Windsor, UK, September 2007.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Techniques for diagnosing software faults. Technical Report TUD-SERG-2008-014, Delft University of Technology, 2008.

[3] J. De Kleer. Getting the probabilities right for measurement selection. In *Proc. DX'06*, Spain, May 2006.

[4] J. De Kleer. Diagnosing intermittent faults. In *Proc. DX'07*, May 2007.

[5] J. De Kleer, A. K. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222, 1992.

[6] J. De Kleer and B. C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.

[7] A. Feldman, G. Provan, and A. J. C. van Gemund. Computing minimal diagnoses by greedy stochastic search. In *Proc. AAAI'08*, Chicago, USA, July 2008.

[8] A. Groce. Error explanation with distance metrics. In *Proc. TACAS*. Springer, 2004.

[9] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Proc. SPIN*. Springer, 2003.

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. ICSE'94*, Sorrento, Italy, 1994.

[11] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *Proc. ISSTA'07*, London, UK, July 2007.

[12] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. ASE'05*, pages 273–282. ACM Press, 2005.

[13] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proc. ESEC/FSE-13*, Lisbon, Portugal, 2005. ACM.

[14] W. Mayer and M. Stumptner. Abstract interpretation of programs for model-based debugging. In *Proc. IJCAI'07*, 2007.

[15] W. Mayer and M. Stumptner. Models and tradeoffs in model-based debugging. In *Proc. DX'07*, May 2007.

[16] B. Peischl, S. Soomro, and F. Wotawa. Abstract dependence models in software debugging. In *Proc. DX'06*, Spain, May 2006.

[17] X. Pucel, S. Bocconi, C. Picardi, D. Dupré, and L. Massuyès. Diagnosability analysis for web services with constraint-based models. In *Proc. DX'07*, May 2007.

[18] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, April 1987.

[19] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proc. ASE'03*, October 2003.

[20] F. Wotawa, M. Stumptner, and W. Mayer. Model-based debugging or how to diagnose programs automatically. In *Proc. IAE/AIE'02*. Springer-Verlag, 2002.

[21] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault localization. In *Proc. AADEBUG*, 2005.