An Observation-based Model for Fault Localization[*]

Rui Abreu Peter Zoeteweij 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

Automatic techniques for helping developers in finding the root causes of software failures are extremely important in the development cycle of software. In this paper we study a dynamic modeling approach to fault localization, which is based on logic reasoning over program traces. We present a simple diagnostic performance model to assess the influence of various parameters, such as test set size and coverage, on the debugging effort required to find the root causes of software failures. The model shows that our approach unambiguously reveals the actual faults, provided that sufficient test cases are available. This optimal diagnostic performance is confirmed by numerical experiments. Furthermore, we present preliminary experiments on the diagnostic capabilities of this approach using the single-fault Siemens benchmark set. We show that, for the Siemens set, the approach presented in this paper yields a better diagnostic ranking than other well-known techniques.

Categories and Subject Descriptors

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

General Terms

Reliability, Experimentation, Measurement.

Keywords

Test data analysis, software fault diagnosis, program spectra, model-based diagnosis.

1. INTRODUCTION

Automated diagnosis of software faults can drastically increase debugging efficiency, improving reliability and time-to-market. In the model-based diagnosis (MBD) domain there have been a number of approaches to automated debugging [13, 17] that reason over a model of the program. Given a compositional, behavioral model of the program under analysis and a set of (real-world) input/output observations, MBD infers which components are likely to explain the differences between the model and the real world. A drawback of this approach is that model extraction is based on *static* program analysis, which limits useful information to compile-time, prohibiting the use of, e.g., (input) data dependent conditional control flow information at run-time, as well as run-time error detection.

Currently, we study a model-based diagnosis approach that uses *dynamic* information to extract a model of program behavior [1], which is inspired by spectrum-based software localization (SFL) approaches [2, 10, 11, 12], which are based on analyzing run-time traces of component activity. As model-based diagnosis traditionally considers multiple faults [4, 5, 6, 7], diagnoses of multiple-fault programs are returned as fault sets. In contrast, SFL (and most of the above automated debugging approaches), just return a single list of *all* individual components, ranked in order of the extent to which their behavior is statistically similar to the occurrence of failures.

For example, consider a triple-fault (sub)program with faulty components c_1, c_2, and c_3. Whereas under ideal testing circumstances our multiple-fault approach would simply produce one single multiple-fault diagnosis $\{\{1, 2, 3\}\}$ (in terms of component indices), an SFL approach would produce multiple single-fault diagnoses like {{1}, {2}, {3}, {4}, {5}, . . .}. Although the higher statistical similarity of the first three items direct the debugging effort in the proper direction, the former, multiple-fault diagnosis unambiguously reveals (1) the actual triple fault, and (2) the fact that exactly three developers can be deployed efficiently (debugging parallelism [9]). Although in practice, such an "optimal" multiple-fault diagnosis cannot always be deduced (due to limited test set size and coverage) this potential optimality is the major rationale behind our multiple-fault approach.

In this paper we provide a more detailed rationale for the above "optimality" of our dynamic, spectrum-based, multiple-fault approach. More specifically,

- We present a diagnostic performance model for a probabilistic program model with parameters such as the number of faults, components, and test runs.
- We show that for any program (model) there exists a test set (spectra) that unambiguously reveals the actual fault (the "optimality" argument).

[*] 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.

WODA – Workshop on Dynamic Analysis, July 21, 2008

$$\begin{array}{cc}
 & \begin{array}{cccc|c} \multicolumn{4}{c}{M \text{ components}} & \text{error vector} \end{array} \\
N \text{ spectra} & \left[\begin{array}{cccc|c}
o_{11} & o_{12} & \cdots & o_{1M} & e_1 \\
o_{21} & o_{22} & \cdots & o_{2M} & e_2 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
o_{N1} & o_{N2} & \cdots & o_{NM} & e_N
\end{array}\right]
\end{array}$$

Figure 1: Observation Matrix O

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

Figure 2: A defective function

- We present initial experiments with our observation-based approach using the Siemens set. As this benchmark only contains programs with single-faults, our approach was only evaluated in this context.

The paper is organized as follows. In the next section we present the concepts and definitions used in this paper. Section 3 presents the basic principles of model-based diagnosis. In Section 4 our observation-based approach for fault diagnosis is described. Our analytical performance model is presented in Section 5. The diagnostic performance on the Siemens set is evaluated in Section 6. We conclude and discuss future work in Section 7.

2. CONCEPTS & DEFINITIONS

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

A program that is being diagnosed comprises a set of M components (statements in the context of this paper), which is executed using N test cases that either pass of fail. Program (component) activity is recorded in terms of program spectra, which are abstractions of program traces.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 was involved in a (test) run or not.

Both spectra and pass/fail information is input to traditional SFL, as well as to our reasoning technique. The combined information is expressed in terms of the $N \times (M + 1)$ *observation matrix* O (see Figure 1). An element o_{ij} is equal to 1 if component j was observed to be *involved* in the execution of run i, and 0 otherwise. The element $o_{i,m+1}$ is equal to 1 if run i *failed*, and 0 if run i *passed*. The rightmost column of O is also denoted as e (the error vector).

For $j \leq M$, 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. From O it is also possible to derive the probability r that a component is actually executed in a run (testing code coverage), and the probability g that a faulty component is actually exhibiting good behavior (testing fault coverage, also known as the "goodness" parameter g from MBD [4]).

Programs can have multiple faults, the number being denoted C (fault cardinality). A *diagnosis candidate* is expressed as the set of indices of those components whose *combined* faulty behavior is logically consistent with the observations O and therefore must be considered as a collective candidate. A *diagnosis* is the ordered set of diagnostic candidates $D = \{d_1, \ldots, d_k\}$, all of which are an explanation consistent with observed program behavior (O), ordered in probability of being the program's actual multiple fault condition. An example multiple-fault diagnosis is the diagnosis $\{d_1\} = \{\{1, 2, 3\}\}$ given in the Introduction. For brevity, we will often refer to diagnostic candidates as diagnoses as well, as it is clear from the context whether we refer to a single diagnosis candidate or to the entire diagnosis.

3. MODEL-BASED DIAGNOSIS

In this section we briefly describe the principles underlying model-based diagnosis as far as relevant to this paper. The purpose of diagnosis is to identify the system components that are the *root cause* of observed failures. We consider a system of M components, that applies some system function $\underline{y} = \mathscr{F}(\underline{x}, \underline{h})$, where $\underline{x}$ and $\underline{y}$ represent observations of system input and output, respectively, and where $\underline{h} = (h_1, \ldots, h_m)$ indicates the *health state* of the system. The health states of a component are healthy and faulty. Diagnosis can be understood as solving the inverse problem $\underline{h} = \mathscr{F}^{-1}(\underline{x}, \underline{y})$, i.e., find the combinations of component health states that explain the observed output for a given input. As the internals of the system are not observable, the diagnosis problem differs from component testing problems.

To put MBD into perspective with respect to this paper, consider the simple program function in Figure 2 which is composed of three inverting statements (with a fault in statement 3), resembling a binary circuit example often used within the model-based community (e.g., see [14]). The function takes one input ($\underline{x} = x$), and returns two outputs ($\underline{y} = (y_1, y_2)$). A *weak* model of each inverter statement is given by the logical proposition

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

which only specifies nominal (required) behavior (a *strong* model would also include a proposition specifying faulty behavior, which requires more modeling / specification effort).

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

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

3.1 Computing Diagnoses

Consider the observation $obs = ((x, y_1, y_2) = (1, 1, 0))$. 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

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

also known as *conflicts* [6], 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* [15], over the above conflicts, 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 due to the weak model.

3.2 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 according to the probability of being the actual fault state.

For each diagnosis candidate the probability that it describes the actual system fault state depends on the extent to which that candidate explains all observations. 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, the prior probability a particular diagnosis d_k is correct is given by $\Pr(d_k) = p^{|d_k|} \cdot (1-p)^{M-|d_k|}$. In order to compute the posterior probability given an observation we use Bayes' rule

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

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

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

In the context of model-based diagnosis, many policies exist for ε [4]. In the above example, we define $\varepsilon = 1/dx$ where dx is the number of observations that can be explained by diagnosis d_k. Returning to the example of Section 3.1, as there are 4 possible observations that can be explained by $\{3\}$, and 8 that can be explained by $\{1, 2\}$, it follows (see [1])

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

Hence, the diagnostic report is as follows

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

Thus the most probable cause of the observed failure is c_3 being faulty. Consequently, debugging would start with the actual faulty statement.

4. OBSERVATION-BASED MODELING

The above approach is dependent on the existence of a model of the program. Even if a model was available for each component (statement), only for the simplest of programs (such as our example programs) a program model could be extracted based on static dependence analysis. However, if our program would involve conditional control flow, compilation into a set of propositions is not straightforward, effectively prohibiting such a model-based reasoning approach. In this section we present our observation-based diagnosis approach, where a model is generated from the observation matrix. 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.

4.1 Computing Diagnoses

Unlike the MBD approaches mentioned earlier, which statically deduce information from the program source, we use O as the *only*, dynamic source of information, from which both a model, and the input-output observations are derived. Apart from the fact that we exploit dynamic information, this approach also allows us to apply a generic component model, avoiding the need for detailed functional modeling, or relying, e.g., on invariants or pragmas for model information.

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 a faulty 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 input and output correctness observation. The overall approach can be viewed as a sequential diagnosis that incrementally takes into account new program (and pass/fail) evidence with increasing N. A single row $O_{n,*}$ corresponds to the (sub)model

$$\begin{aligned} h_m &\Rightarrow (x_m \Rightarrow y_m),\ \text{for } m \in S_n \\ x_{s_i} &= y_{s_{i-1}},\ \text{for } i \geq 2 \\ x_{s_1} &= \text{true} \\ y_{s'} &= \neg e_n \end{aligned}$$

where $S_n = \{m \in \{1, \ldots, M\} \mid o_{nm} = 1\}$ denotes the well-ordered set of component indices involved in computation n, s_i denotes the i^{th} element in this ordering, (i.e., for $i \leq j, s_i \leq s_j$), and s' denotes its last element. The resulting component chain logically reduces to

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

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

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

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

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

As in MBD, 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 $\neg h_1$ and $\neg h_3$. Compared to the MBD approach, the second conflict ($\neg h_2 \vee \neg h_3$) is missing due to the fact that no knowledge is available on component behavior and component interconnection. Although this suggests that the dynamic approach yields lower diagnostic performance, note that the example program does not have conditional control flow, and hence is ideally suitable to static analysis.

4.2 Ranking Diagnoses

The probability computation is generally based on the MBD approach as explained in the previous section. Although for each component the a priori fault probability $\Pr(\{j\})$ is typically dependent on code complexity, design, etc., we will simply assume $\Pr(\{j\}) = p$ ($p = 0.01$ in the context of this paper). Again, assuming components fail independently, the prior probability a particular diagnosis d_k is correct is given by $\Pr(d_k) = p^{|d_k|} \cdot (1-p)^{M-|d_k|}$. 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 3.2

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

where $\Pr(obs|d_k)$ is defined as

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

As in software components it is quite usual that a faulty component exhibits correct behavior, we use the "goodness" parameter g introduced earlier in the ε strategy, according to

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

where t is the number of faulty components (according to d_k involved in the run (the rationale being that the more faulty components are involved, the more likely it is that the run will fail), and where g is estimated by [4]

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

where $[\cdot]$ is Iverson's operator ($[\text{true}] = 1$, $[\text{false}] = 0$]). Note that the hitting set computation is inherently considered by the 0-clause in the probability update. As the hitting set can be determined first, the probability computation need only be performed on the diagnostic candidates contained in this set.

Similar to the model-based diagnosis approach presented in Section 3, c_3 is also more probable to be the root cause of the failure ($\Pr(\{3\}) = 0.8$) because it has only been executed in a failed computation (whereas c_1 was also involved in a passed computation, which exonerates it from being at fault, having $\Pr(\{1\}) = 0.2$).

5. ANALYTIC MODEL

In this section we derive a simple, approximate model to assess the influence of various parameters on the *wasted* debugging effort W. It is defined as the effort that is wasted on inspecting a component that was not faulty. 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 didn't change the second candidate). In contrast to related work, we measure W instead of effort so that the performance metric's scale is independent of the number of faults in the program.

The evaluated parameters are number of components M, number of test cases N, testing code coverage r, testing fault coverage g, and fault cardinality C. Consider the example O in Figure 3(a), with $M = 5$ components of which the first $C = 2$ components are faulty. As a faulty component can still produce correct behavior which does not not cause a run to fail, we use an extended encoding where '1' denotes a component that is involved, and ‘2’ denotes a (faulty) component whose involvement actually produced a failure (and consequently a failing run).

c_1	c_2	c_3	c_4	c_5	e
1	0	1	0	1	0
0	2	1	0	0	1
0	2	1	1	0	1
1	1	1	1	0	0
2	1	0	1	0	1

(a) Example O

c_1	c_2	c_3	c_4	c_5	e
0	2	1	0	0	1
0	2	1	1	0	1
2	1	0	1	0	1

(b) O’s failed runs only

Figure 3: Observation Matrix Example

In the following we focus on the hitting set since its constituents are primarily responsible for the asymptotic behavior of W. Although their individual ranking is influenced by component activity in passed runs, the hitting set itself is exclusively determined by the failing runs. Thus, we consider the sub-matrix shown in Figure 3(b).

From Figure 3(b) it can be seen that the first 2 columns together form a hitting set of cardinality 2 (which corresponds to our choice $C = 2$). This can be seen by the fact that in each row there is at least one set member involved, i.e., there is a so-called “chain” of c_1 and/or c_2 involvement that is “unbroken” from top row to bottom row.

While this chain exists by definition (given the fact that both are faulty there is always at least one of them involved in *every* failed run), other chains may also exist, and may cause W to increase. This occurs when those chains pertain to diagnostic candidates of equal or lower cardinality (B) than C. Generally, two types of chain can be distinguished: (1) chains (of cardinality $B < C$) within the faulty components set, called *internal* chains, and (2) chains (of cardinality $B \leq C$) completely outside the faulty components set, called *external* chains. In the above example after $N = 2$ (so considering only the first two failed runs), there is still one internal chain (corresponding to single fault c_2), and two external chains (corresponding to single fault $\{3\}$, and double fault $\{3, 4\}$). As their probability will be higher (due to the a priori probability computation) they will head the ranking. With respect to the internal fault this does not significantly influence W since this indicates a true faulty component (the real double fault $\{1, 2\}$ being subsumed by $\{2\}$). Consequently, there is no wasted debugging effort. With respect to $\{3\}$ however, this fault will induce wasted effort. After $N = 3$ both single faults has disappeared (both chain of ‘1’s have been *broken* during the third failing run), while the double fault c_3, c_4 is still present. From the above example it follows that (1) W is primarily impacted by external chains, and (2) the probability of a B cardinality chain still “surviving” decreases with the number of failing runs. The latter is the reason why in the limit for $N \to \infty$ all external (and internal) chains will have disappeared, exposing the true fault as only diagnosis.

5.1 Number of Failing Runs

As the number of failing runs is key to the behavior of W, in the following we first compute the fraction of failed runs f out of the total of N runs, given r and g. Consider C faulty components. Let f denote the probability of a run failing. A run passes when none of the C components induces a failure, i.e., does not generate a ‘2’ in the matrix. Since the probability of the latter equals $1 - r \cdot (1 - g)$ and generating a ‘2’ requires (1) being involved (probability r) and (2) producing a failure (probability $(1 - g)$), the probability of not generating a ‘2’ in the matrix equals $(1 - r \cdot (1 - g))$. Consequently, the probability a run passes equals $(1 - r \cdot (1 - g))^C$, yielding

$$f = 1 - (1 - r \cdot (1 - g))^C$$

This implies that for high g (and/or low r) a very large number of runs N is required to generate a sufficient number $N_F = f \cdot N$ of failing runs in order to eliminate competing chains of equal of lower cardinality B. As r also affects the number of external chains which, however, is not affected by g, the effect of g can be seen orthogonal to r in that it only impacts the number of failed runs through f. Consequently, g and N are related in that a high g is compensated by a, possible huge, increase in N. In the sequel, we therefore only focus on the effect of r.

5.2 Behavior for Small Number of Runs

While for large N the determination of W depends on the probability that competing chains will have terminated, for small N a more simple derivation can be made. Consider the case of a single failing run ($N_F = f \cdot N = 1$). From the first (failing) row ($k = 1$) in the above example (Figure 3(b)) it can be seen that there are generally $r \cdot (M - C)$ external single-fault ($B = 1$) chains (c_3 and c_5) that induce wasted effort. As W denotes the ratio of wasted effort it follows

$$W = \frac{r \cdot (M - C)}{M} \tag{1}$$

which for large M approaches r. This is confirmed by the experiments discussed later.

After the second failed run ($k = 2$) the probability a $B = 1$ chain survives two failing runs equals r^2 (i.e., the probability of two ‘1’s for a particular component). Consequently, the number of $B = 1$ chains equals $r^2 \cdot (M - C)$, which, in general, decreases negative-exponentially with the number of (failing) runs ($f \cdot N$). For $B = 2$ the situation is less restrictive as *any* combination of ‘1’s of the first and second row qualifies as a double-fault chain. As on average there are $M' = \lfloor r \cdot (M - C) \rfloor$ ‘1’s per row there are $\binom{M'}{2}$ double-faults.

After the third failing run ($k = 3$) the number of surviving $B = 1$ chains equals $r^3 \cdot (M - C)$, whereas the number of triple faults equals $\binom{M'}{3}$. As for sufficiently large M the higher-cardinality combinations outnumber the lower-cardinality combinations, W is dominated by the combinations that have the same cardinality as the fault cardinality C. Consequently, assuming $N_F \leq C$ it follows that the number of C-cardinality chains that compete with the actual C-cardinality diagnosis is approximated by $\binom{M'}{C}$. However, if there are more combinations than $M - C$ these combinations will overlap in terms of component indices. As W does not measure wasted effort on a component that was already previously inspected (and subsequently removed from the next diagnosis), the average number of “effective” C-cardinality chains will never exceed $\frac{M}{C}$ (as there are C indices per candidate). Hence, the number of competing C-cardinality chains is approximated by $\min(\frac{M}{C}, \binom{M'}{C})$.

5.3 Behavior for Large Number of Runs

For large N_F the trend of W can also be approximated from the probability that competing chains will still have survived after N_F runs, which we derive as follows. Consider a B-cardinality external chain. At each row there is a probability that this chain does not survive. Similar to the derivation of f we consider the probability that *all* B components involved in the chain have a '0' entry, which would terminate that particular chain. This probability equals $(1-r)^B$. Hence, the probability that a B-cardinality chain does not break per run equals $1-(1-r)^B$. Consequently, the probability that a chain survives N_F failing runs equals

$$(1-(1-r)^B)^{N_F}$$

Similar to the derivation for small N_F, we only consider C-cardinality chains. The largest number of competing chains at the outset equals $\binom{M'}{C}$. As there always exists an N_F for which this number is less than $\frac{M}{C}$ (in the asymptotic case we consider only a few chains) the number of competing chains after N_F runs is given by

$$(1-(1-r)^C)^{N_F} \cdot \binom{M'}{C}$$

Consequently, W is approximated by

$$W \approx \frac{(1-(1-r)^C)^{N_F} \cdot \binom{M'}{C}}{M} \tag{2}$$

We observe a negative-exponential (geometric) trend with N_F (N) while C postpones that decay to larger N_F (N) as the term $1-(1-r)^C$ approaches unity for large C.

In the following we asymptotically approximate the number of failing test runs N_F needed for an optimal diagnosis (i.e., W approaches 0). Considering Eq. (2) a single diagnosis is approximately reached for

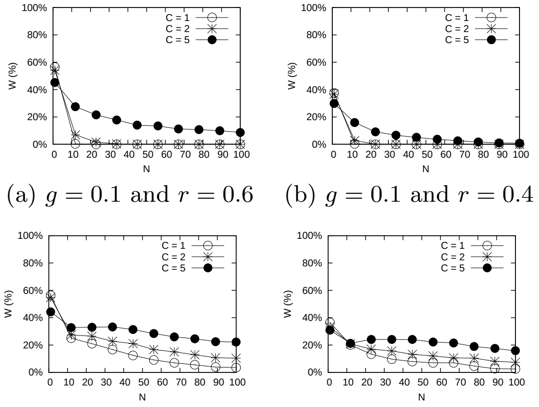
$$(1-(1-r)^C)^{N_F} \cdot \binom{M'}{C} = W \cdot M$$

which can be modeled as $(1-(1-r)^C)^{N_F} = K$. It follows $N_F = -\log K / \log 1-(1-r)^C$. Since for sufficiently large C the term $1-(1-r)^C$ approaches unity, and since $\log 1-\epsilon \approx -\epsilon$ it follows that $N_F \sim \log K/(1-r)^C$. As $(1-r) < 1$ it follows $N_F \sim \log K \cdot ((1-r)^{-1})^C$ of which the second term increases exponentially with C. Since $K = \binom{M'}{C}$ for large M this term also increases exponentially with C. However, as the term is included in a logarithm, the effect of this term is less than the previous. In the next section we numerically verify the exponential trend of N.

5.4 Experimental Validation

In this section we experimentally validate the predictions of the model just outlined. For that purpose, an observation matrix generator was implemented, which takes into account the following parameters: N, r, g, and C.

The graphs in Figure 4 plot W versus N, for $C = 1$, $C = 2$ and $C = 5$, respectively. Each measurement represents an average over 1,000 sample matrices with $M = 20$ (see [1] for plots with other M, r, and g values). The plots show that W for $N = 1$ is similar to r as predicted in Eq. (1), while for sufficiently large N all techniques produce an optimal diagnosis. Besides, 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 $C = 1$, 10 runs would be enough for a perfect diagnosis, whereas for $C = 5$, 250 runs would be needed. For small g almost each run that involves the faulty component yields a failure ($f \approx 1$), already producing near-perfect diagnoses for only small N. For high g the transition between the small-N_F behavior and large-N_F behavior is visible. As the negative-exponential trend with N_F is clear from the analytical model 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. 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.



(a) $g = 0.1$ and $r = 0.6$ (b) $g = 0.1$ and $r = 0.4$

(c) $g = 0.9$ and $r = 0.6$ (d) $g = 0.9$ and $r = 0.4$

Figure 4: W vs. N

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 ($r = 0.6$)

6. EXPERIMENTAL EVALUATION

In this section we report our first experiments with the Siemens set, which only contains programs with single faults.

6.1 Experimental Setup

To study the diagnostic performance of the observation-based modeling approach, we use a set of programs widely used in the field, called the Siemens set [8]. It is composed of seven different programs. Every single program has a correct version and a set of faulty versions of the same program. Each faulty version contains exactly one fault. Each program also has a set of inputs that ensures full code coverage. Due to space limitations we do not provide the details of the programs (for detailed information see [8]). In total the Siemens set provides 132 programs. However, as no failures are observed in two of these programs, namely version 9 of `schedule2` and version 32 of `replace`, they are discarded. Besides, we also discard versions 4 and 6 of `print_tokens` because the faults in this versions are global variables in an header file and the profiling tool (`GNU gcov`) used in our experiments does not log the execution of these statements. In summary, we discarded 4 versions out of 132 provided by the suite, using 128 versions in our experiments.

For compatibility with previous work in fault localization, we use the effort/$s$core metric [2, 12] which is the percentage of statements that need to be inspected to find the fault - in other words, the rank position of the faulty statement divided by the total number of statements. Note that some techniques such as in [12] do not rank all statements in the code, and their rankings/effort are therefore based on the program dependence graph (PDG) of the program [16].

6.2 Experimental Results

Table 2 presents the cumulative percentage of faults found when a certain debugging effort is spent. The results for observation-based modeling (ObM) were obtained by limiting the (hitting) set of valid diagnoses to single-fault explanations. The values for Sober [12] (a similar technique is presented in [11], which is not included in our comparison because Sober consistently outperforms it [12]), Tarantula [10], and Ochiai [2] were obtained by running them in our own environment, those for Cause Transitions (CT) are, however, directly cited from [3].

Clearly, the ObM approach consistently outperforms all other techniques, requiring less effort to find more faults. For example, the ObM approach was able to assist the programmer in finding the faulty location for 18% of the faulty versions (i.e., roughly 23 versions) by examining less than one percent of the source code. This represents an improvement over Ochiai of about 4% (6 versions), 6% over Tarantula (8 versions), 10% over Sober and 13% over CT. If 10% of the code would be inspected, ObM would lead to find 60% of the faults, whereas only 49% would be found with Sober. Note that CT is always the worst performing technique. However, it relies only on two runs to explain the failure, whereas our approach needs several failed runs to compute a meaningful minimal hitting set.

Effort	Tarantula	Ochiai	Sober	CT	ObM
< 1	12	14	8	5	18
< 10	46	52	49	26	60
< 20	60	65	65	38	71
< 30	72	78	72	51	81
< 40	79	80	72	53	83
< 50	80	83	81	60	93
< 60	90	96	84	63	99
< 70	96	99	85	71	100
< 80	98	100	90	75	100
< 90	100	100	92	82	100
≤ 100	100	100	100	100	100

Table 2: Cumulative percentage of faults found

Although initial experiments seem to suggest that the approach can be of added value to help developers pinpointing the root causes of different failures (i.e., programs with multiple faults), we refrain from generalizing (and comparing with other techniques) because more experimentation is required.

7. CONCLUSIONS AND FUTURE WORK

In this paper we studied a model-based diagnosis approach which uses dynamic information, namely abstraction of program traces, to generate a (dynamic, sub-) model of the program under analysis. The model, along with the set of traces for pass/fail executions is used to reason about the observed failures. In contrast to most approaches to software fault diagnosis, which present diagnosis candidates as single explanations [2, 10, 12], our approach also contains multiple fault explanations in the diagnostic ranking (typical of model-based approaches [13, 17]).

We have analytically and empirically demonstrated that the observation-based approach will reveal the true faulty state of the program as an unambiguous, multiple-fault diagnosis explanation for failures, given sufficient test cases are provided. The analytical model also reasons about the influence of several parameters on the diagnostic accuracy of the approach. Studied parameters are number of faults, components, and test runs.

Furthermore, we applied the approach to the widely-used Siemens set of programs, which is a set containing programs with single-faults. In this context, our approach clearly outperforms several other well-known techniques.

Future work includes the following. As an intrinsic characteristic of model-based diagnosis approaches, the one in this paper generally finds multiple-fault diagnosis candidates as possible solutions. Therefore, we plan to study the diagnostic performance of our approach in such context for real programs, as well as the possibility of exploiting such information to efficiently engage several developers to repair the defect(s) in parallel. Furthermore, we plan to implement a statistics-directed approach for reducing the time complexity of the hitting set computation, allowing a fair (time) comparison with other approaches. Finally, we intend to study the impact of real program characteristics on our approach, such as test suite size and defect location/type.

8. ACKNOWLEDGMENTS

We extend our gratitude to Johan de Kleer for discussions which have influenced our reasoning approach. Many thanks go to the anonymous reviewers for their useful comments.

9. REFERENCES

[1] 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.

[2] 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. IEEE CS.

[3] H. Cleve and A. Zeller. Locating causes of program failures. In *Proc. ICSE'05*, St. Louis, Missouri, USA, May 2005.

[4] J. de Kleer. Diagnosing intermittent faults. In *Proc. DX'07*, Nashville, TN, USA, 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. *Artificial Intelligence*, 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, Illinois, USA, July 2008.

[8] 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.

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

[10] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. ICSE'02*, Orlando, Florida, USA, May 2002.

[11] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. PLDI'05*, Chicago, IL, USA, 2005.

[12] 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.

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

[14] J. Pietersma and A. J. C. van Gemund. Temporal versus spatial observability tradeoffs in model-based diagnosis. In *Proc. SMC'06*, Taipei, Taiwan, October 2006. IEEE CS.

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

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

[17] F. Wotawa, M. Stumptner, and W. Mayer. Model-based debugging or how to diagnose programs automatically. In *Proc. IAE/AIE'02*, volume 2358 of *LNCS*. Springer-Verlag.