Model-Based Software Debugging

Rafael Ceballos, Rui Abreu, Ángel Jesús Varela-Vaca and Rafael M. Gasca

15.1 Introduction

The complexity and size of software systems have rapidly increased in recent years, with software engineers facing ever-growing challenges in building and maintaining such systems. In particular, testing and debugging, that is, finding, isolating, and eliminating defects in software systems still constitute a major challenge in practice [47].

Debugging is an iterative process, where hypothesis generation, hypothesis selection, and hypothesis confirmation form the central tasks [7]. However, selection and exploration of good hypotheses remain difficult, and programmers often rely on intuition and spend considerable effort on pursuing seemingly promising hypotheses that ultimately do not lead to the true fault [37]. Similar to physicians, who often apply multiple tests to arrive at a diagnosis, software engineers have multiple debugging tools to analyze a program failure at their hands. In the software domain, debugging tools can leverage information obtained from (i) the execution of the program and from (ii) formal analysis of the behavior of a program or its model. While a rich set of tools has been developed to ease the burden of gathering information about a program and its execution(s), complementary approaches to use and analyze this

R. Ceballos (✉) · Á. J. Varela-Vaca · R. M. Gasca
University of Seville, Seville, Spain
e-mail: ceball@us.es

Á. J. Varela-Vaca
e-mail: ajvarela@us.es

R. M. Gasca
e-mail: gasca@us.es

R. Abreu
IST, University of Lisbon and INESC-ID, Lisbon, Portugal
e-mail: rui@computer.org

information must be leveraged to target a broader spectrum of faults and compensate the limitations of individual techniques.

Model-Based Software Debugging (MBSD) is a technique that leverages concepts from program slicing. In MBSD, a diagnosis is obtained by the logical inference from the static model of the system, combined with a set of run-time observations. There are several types of models, derived from the source code and test cases that are then used by the technique to reason about observed failures. Despite the accuracy of this technique, in most cases, the computational effort required to create a model of a large program forbids the use of model-based approaches in real-life applications [29, 40, 42–44, 52]. Regardless of the complexity of the approach, there have been developments into making the real-life applications more tractable to model-based debugging:

- One option is the combination of MBSD with a more lightweight and accurate technique that uses coverage to reason about observed failures [40] Yet another field of application is end-user programming, such as spreadsheets, where programs tend to be smaller [31].
- Another option is based on the combination of different paradigms, such as Design by Contract. This methodology is named Software Diagnosis Based on Constraints Models [20, 54]. The goal is to isolate and identify faults in assertions (Design by Contract) and/or in sentences (source code). Design by Contract was proposed in [41], in order to improve the Object-Oriented software quality.

The remainder of this section elaborates on techniques to isolate software faults, in particular, model-based approaches.

15.2 Background and Problem Statement

Improving software quality has been a long-standing issue in academia, leading to considerable advances in automated fault detection, localization, and correction. While fault detection strives to expose defects in a given program, fault localization aims to identify particular program fragments that may be responsible for a failure, and fault correction aims to repair the program.

Effective (automated) debugging assistance can be provided by supporting the developer in one or more of the aforementioned tasks. Recent advances in program analysis and verification techniques have led to mature mathematical frameworks and tools focused on specific purposes, but their individual utility for general debugging remains limited to specific programs, execution environments, and problem contexts. As a result, the overall debugging process remains complex and much time is devoted to analyzing (possibly irrelevant) results gathered from different tools.

For example, techniques that are solely based on a program, such as slicing [49] or invariant learning [28], are often ineffective in locating faults that stem from functionality that has been “forgotten” in the implementation or that is not covered

by tests. Conversely, techniques based on abstract specifications (if available), such as abstract state machines [56], can locate regions in a program where specification and program show different behaviors but may suffer from difficulties obtaining sufficiently detailed, correct specifications that would allow to confine the cause to a specific small region. Only by a combination of complementary techniques can such complex faults be located effectively.

However, many formal methods suffer from spurious explanations that are caused by approximations and abstractions introduced by the formalism [13], while others may return results that are too large to be useful [46]. Experience with automated debugging and checking tools shows that the remaining results are often dismissed if the fault has not been located after examining the first few candidate explanations [36]. Therefore, discriminating between true explanations and those that are caused by approximations in the analysis is essential. Statistics-based techniques are among the more popular automated fault localization techniques. By correlating information about which program fragments have been exercised in a set of multiple program execution traces (also called program spectra) with information about successful and failing executions, Spectrum-based Fault Localization (SFL) and other statistics-based approaches yield a list of suspect program fragments sorted by their likelihood to be at fault.

Since this technique is efficient in practice, this and other dynamic analysis techniques are attractive for large modern software systems [52]. Overall, statistical techniques are lightweight, but are rather dependent on the availability of a suitable test suite.

Machine learning techniques also feature prominently among automated program analysis tools. In this context, learning is applied to infer from execution traces models that describe the program's intended behavior. For example, Daikon [26] is an invariant detection tool built with the intention of supporting program evolution by helping programmers to better understand the code. It analyzes the values of variables encountered in executions of the program and retains only those Boolean expression over program variables that are satisfied in all executions. For example, if in all observed executions, the value of variable x was less than a variable y at some point in the execution, the invariant $x < y$ will be reported. The same approach has since been applied to debugging, where violations of inferred invariants are used to detect errors [28]. However, existing work does little to help determine the origin of the fault once a failure has been detected, and the learning algorithms may produce results that are too numerous or too specific to be of value to the programmer.

Better results than those obtained from methods based on dynamic analysis alone can often be achieved if a model of the correct program behavior is available [1]. Model-Based Software Debugging (MBSD) techniques have been advocated as powerful debugging aid that isolate faults in complex programs [40]. By comparing the execution 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 fully explain the observed symptoms. Compared to spectrum-based localization, model-based analysis yields better accuracy due to precise reasoning about the possible effects of each program fragment but suffers from poor scalability.

Better and precise results can be achieved if the model-based methodology for diagnosing bugs takes into account Design by Contract (DbC) methodology because the correct behavior is available for automatic reasoning [21]. DbC is the model of the correct behavior that the source code must satisfy. In [50], two measures in order to validate the benefits of using DbC are proposed: robustness and diagnosability. The robustness is the degree to which the software is able to recover from internal faults that would otherwise have provoked a failure. DbC enables the development of more reliable and robust software applications and the control of abnormal situations. Diagnosability expresses the effort required in the localization of a fault as well as the precision allowed by a test strategy on a given system. The results show that the robustness improves rapidly with only a few contracts, and for improving diagnosability the quantity of contracts is less important than their quality. In [14], it is shown that contracts are useful for fault isolation if they are defined during analysis. By using contracts, the fault isolation and diagnosability are significantly improved in object-oriented code (it implies a wider distribution of functions).

In this chapter, we use the terminology introduced by Avizienis et al. [8]:

- A *failure* is an event that occurs when delivered service deviates from correct service.
- An *error* is a system state that may cause a failure.
- A *fault* (defect/bug) is the cause of an error in the system.

In the context of this chapter, faults are bugs in the code of a software program.

Definition 15.1 A software program Π is formed by a sequence M of one or more *components* (e.g., statements). Components can be of several levels of granularities, such as classes, methods, and statements.

Failures and errors are symptoms caused by faults in the program. Fault localization aims at isolating the root cause of observed symptoms. The fault localization techniques considered in this chapter consider the existence of a test suite revealing the software faults.

Definition 15.2 A test case t is a (i, o) tuple, where i is a collection of input settings or variables for determining whether a software system works as expected or not, and o is the expected output. If $\Pi(i) = o$ the test case passes, otherwise fails.

Definition 15.3 A test suite $T = \{t_1, \ldots, t_N\}$ is a collection of test cases that are intended to test whether the program follows the specified set of requirements. The cardinality of T is the number of test cases in the set $|T| = N$.

15.3 Software Diagnosis Based on Constraints

A methodology based on constraints for diagnosing software is proposed in [21]. The main idea is the transformation of contracts and source code into a model based on constraints for locating faults or defects in source code and assertions. These faults

are wrongly designed assertion or statements, for example, variations in Boolean conditions or in assignment statements. Other types of errors, as syntax errors, memory access violations, or infinite loops, could be considered in future work.

15.3.1 Constraints-Based Model

A diagnosis is a hypothesis about what are the changes to do in a program in order to obtain a correct behavior. A component has an abnormal behavior [35] if the outputs are different from the expected results. For example, a multiplier component is abnormal if the output of the multiplier is different to the multiplication of its inputs. For each component c, a Boolean variable $AB(c)$ stores if c is abnormal or not, in order to know whether a component is abnormal and can be a part of the minimal diagnosis. The goal of this methodology is detecting semantic defects in source code or assertions, and these defects are modeled as components with abnormal behavior.

In a software program, blocks of source code are linked in order to obtain the specified behavior. Each statement of a source code can be considered as a component, with inputs and outputs (results). An executed program is a set of linked blocks of code. The order of these blocks can be represented as a *Control Flow Graph* (CFG). The CFG is a directed graph that represents a set of sequential blocks and decision statements. A *Path* is a possible sequence of statements of the CFG. In order to detect and isolate defects in the design of programs, the CFG and program contracts are transformed into a model based on constraints. Testing techniques select the observational models which are most significant for detecting failures in programs. A test case is designed for executing a particular program path and determining whether a program works as expected or not.

When a program has executed the order of the assertions and statements is very important. It is necessary to maintain this order when the source code and contracts are transformed into constraints. For this reason, the statements of the CFG are translated into a Static Single Assignment (SSA) form. This translation maintains the execution sequence when the program is transformed into constraints. In SSA form, only one assignment is allowed for each variable in the whole program. For example, the statements $b = x*q;...b = b+3;...$ $\{Post{:}b = ...\}$ is transformed to $b1 = x1*q1;...b2 = b1 + 3;...\{Post{:}b2 = ...\}$.

The System Model (SM) is a finite set of constraints which determine the software behavior. It will be obtained by transforming the set of statements and assertions of a program to constraints (in SSA form). A test case is an Observational Model that can be applied to an SM.

The subset $D \subseteq SM$ is a diagnosis if $SM' \cup TC$ is satisfied, where $SM' = SM - D$. The minimal diagnoses imply to modify the smallest number of program statements or assertions. A diagnosis is a set of components with abnormal behavior and the goal is to minimize this set. In order to maximize the number of components with normal behavior, the idea is solving a MAX-CSP (CSP and MAX-CSP had been introduced in Chap. 14).

```
/**
 *  @inv getBalance()>= 0
 *  @inv getInterest >= 0
 */
public interface Account {

    /**
     *  @pre income > 0
     *  @post getBalance() >= 0
     */
    public void deposit (double income);

    /**
     *  @pre withdrawal > 0
     *  @post getBalance() ==
     *      getBalance()@pre - withdrawal
     */
    public void withdraw (double withdrawal);

    public double getBalance ();

}
```

```
public class AccountImp implements Account {

    private double interest;

    private double balance;

    public AccountImp()  {
        this.balance = 0;
    }

    public void deposit (double income) {
        this.balance = this.balance + income;
    }

    public void withdraw (double withdrawal) {
        this.balance = this.balance - withdrawal;
    }

    public double getBalance() {
        return this.balance;
    }
}
```

Fig. 15.1 Contract and source code for the bank account class example

The first goal is detecting the inconsistencies between test cases and contracts, and then between test cases, contracts and source code. These inconsistencies are detected, for example, if the SM does not satisfy a Test Case. If there are inconsistencies, the second goal is isolating the inconsistencies between test cases and contracts, and then between test cases, contracts, and source code. These are explained in more detail in the sections below.

15.3.2 Diagnosing DbC Defects

In Fig. 15.1, the class *AccountImp* is shown. It is an example that simulates a bank account. There are methods for depositing and withdrawing money. Assertions are checked in two different ways: first without test cases, and then with test cases.

- Without test cases. Two kinds of checks are proposed:
  - Checking if all the invariants of a class can be satisfied together.
  - Checking if the precondition and postcondition of a method are feasible with the invariants of its class.
- With test cases. The idea is applying test cases to the sequence {invariants + precondition + postcondition + invariants} for each method.

Example 1 In Fig. 15.2, the checking of method *withdraw* is shown. The initial balance must be 0 units and, when a nonnegative amount is withdrawn, the balance must preserve the value 0. The balance must be equal or greater than zero when the method finishes because of the invariant, but the postcondition implies that *balance* = *balance@pre* – *withdrawal*, that is, $0 - withdrawal > 0$, and this is impossible if the *withdrawal* is positive. There is a problem with the precondition since it is not

DbC:	*CSP*:	*Max-CSP*:
Inv.: balance >= 0 Pre.: withdrawal > 0 Post.: balance = balance@pre - withdrawal Inv.: balance >= 0	C_1: balance@pre >= 0 C_2: withdrawal > 0 C_3: balance = balance@pre - withdrawal C_4: balance >= 0 Var = {balance@pre, withdrawal,balance} Dom = {0,[0, 100] , 0}	C_1: AB(Inv) $\vee$ (balance@pre >= 0) C_2: AB(Pre) $\vee$ (withdrawal > 0) C_3: AB(Post) $\vee$ (balance = balance@pre - withdrawal) C_4: AB(Inv) $\vee$ (balance >= 0) Var = {balance@pre,withdrawal, balance,AB(Inv),AB(Pre),AB(Post)} Dom = {0,[0, 100],0,[$false$,$true$], [$false$,$true$],...,[$false$,$true$]}
Test case: Inputs={balance@pre = 0, withdrawal > 0} Outputs= {balance = 0} Source code: Method Withdraw		

Fig. 15.2 CSP for detecting and isolating defects by using a test case for method *Withdraw*

strong enough to stop the program execution when the withdrawal is not equal or greater than the balance of the account.

15.3.3 Diagnosing Source Code Defects

After checking DbC, then the source code of the program is checked by using test cases. A System Model is obtained by transforming DbC assertions (preconditions, postconditions, invariants) and statements. DbC assertions are directly transformed into constraints (into SSA form). The source code outputs must satisfy these constraints because they correspond to the correct behavior.

In order to transform the statements to constraints (System Model), the source code is divided into basic blocks: sequential blocks (declarations, assignments, and method calls), conditional blocks, and loop blocks. For example, an assignment *Ident* = *Exp*; is transformed into an equality constraint in a CSP, and for the MAX-CSP the abnormal behavior is added: AB(S_{Asig}) $\vee$ *Ident* = *Exp*. After execution of S_{Asig}, the equality between the assigned variable and assigned expression must be satisfied, or in another case, this statement (assignment) has an abnormal behavior (the statement contains a defect).

In Fig. 15.3, the polybox example is transformed into five statements (source code). Polybox example had been introduced in Chap. 2. The program cannot reach the correct output because the second statement is an adder (bug) instead of a multiplier. By transforming the source code, a CSP and Max-CSP are obtained. For checking if there are failures, a test case is applied to the CSP. The test case does not satisfy the CSP. There is almost one semantic defect that generates an unexpected output. In order to identify this defect, the same test case is applied to the Max-CSP. By solving this Max-CSP, the obtained minimal diagnosis is $\{S_2\}$. The diagnosis is minimal because the goal of the MAX-CSP is to find an assignment of the AB variables that enable the maximum number of components with normal behavior.

Source Code:	*CSP*:	*Max-CSP*:
S_1: int x = a * c	C_1: x = a × c	C_1: AB(S_1) ∨ (x = a × c)
S_2: int y = b + d	C_2: y = b + d	C_2: AB(S_2) ∨ (y = b + d)
S_3: int z = c * e	C_3: z = c × e	C_3: AB(S_3) ∨ (z = c × e)
S_4: int f = x + y	C_4: f = x + y	C_4: AB(S_4) ∨ (f = x + y)
S_5: int g = y + z	C_5: g = y + z	C_5: AB(S_5) ∨ (g = y + z)
	Var = {a,b,c,d,e,f,g,x,y,z}	Var = {a,b,c,d,e,f,g,x,y,z,AB(S_1),AB(S_2),..., AB(S_5)}
	Dom = {2,3,3,2,2,12,12, $[0,100],[0,100],[0,100]$}	Dom = {2,3,3,2,2,12,12,$[0,100],[0,100],[0,100]$, $[false,true],[false,true],...,[false,true]$}
Test case: Inputs={a = 3, b = 2, c = 2, d = 3, e= 3} Outputs= {f = 12, g = 12} Source code: S1 .. S5		

Fig. 15.3 CSP for detecting and isolating defects by using a test case for the Toy Program.

Example 1	Example 2
{Pre: $x>0 \wedge y>0$}	{Pre: $i>=0 \wedge i<=n \wedge p>0$}
public int dif(int x,int y){	public int rec(int n,
int max,min,s,z;	int i,int p){
(S_1) if (x>=y){	int s;
(S_2) min=y;	(S_1) if (i==n)
(S_3) max=x;	(S_2) s=1;
}else{	else{
(S_4) min=x;	(S_3) p=2*p;
(S_5) max=y;}	(S_4) s=this.rec(n,i+1,p);
{Assert: *max >= min* }	(S_5) s=s+p;}
(S_6) z=max-min;	(S_6) return s; }
(S_7) s=0;	{Post:$s = 1 + p\sum_{y=1}^{n-i} 2^y$}
(S_8) while (z>0){	
(S_9) s=s+z;	
(S_{10}) z=z-1;}	
(S_{11}) return s;	
{Post: $s = \sum_{a=1}^{\lvert x-y \rvert} a$}	

Fig. 15.4 Source code examples that include conditional statements, loops, and method calls

In Fig. 15.4 two more examples are shown. For conditional statements, as in example 1, the predicate *P* applied to each statement store if each statement is part of the *path* or not. When we have a statement S_x, if $P(S_x)$ is true then S_x belongs to the *path* of the executed program; otherwise, it does not belong to the path. For a conditional statement [22], generated constraints include the transformation of the condition and the block of statements included in each case. Only one path is possible, and this path will depend on the condition. The transformation to constraints of the conditional statement allows to detect and isolate defects in conditions and to set the correct path for obtaining the correct behavior. A loop statement can be transformed into a sequence of conditional statements. The number of iterations depends on the test case. For each iteration, the equivalent nested conditional statement is transformed into constraints. Maintaining the order is important for obtaining the same result as

the loop statement. The invariant of the loop is also transformed into constraints of the model.

For method calls and return statements, as in example 2 (Fig. 15.4), is possible to substitute the method call by the precondition and postcondition of the called method. The constraints obtained from the postcondition give us information about the correct behavior. If there is no contract, another option is to substitute the method call by the statements of the method [23].

15.4 Spectrum-Based Reasoning for Software Debugging

This section introduces a lightweight, reasoning technique that reasons over abstractions of program traces, called program spectra, to produce a diagnostic report for observed failures in program executions. The technique is known as Spectrum-based Fault Localization (SFL, for short), and is among the best fault localization techniques [42, 52].

15.4.1 Program Spectra

A program spectrum, introduced by Reps et al. in 1997 to address the Year 2000 problem[1] [32], is a characterization of the execution of a program execution on a set of inputs. These set of inputs can be, for instance, test cases in a test suite. Note that in the following execution, transaction, and test case are used interchangeably.

Program spectra are information collected at run-time, hence it provides a view on the dynamic behavior of a program. A program spectrum is represented as a vector of M counters or flags, where M is the number of *software components*. Various different program spectra exist [30]; e.g., path-hit spectra, data-dependence-hit spectra, and block-hit spectra are among the most common ones. The spectra commonly used in spectrum-based fault localization is component-hit spectra, a type of spectra that merely indicates whether a component was involved in program execution.

Simlar to code coverage tools [55], the source code needs to be instrumented to collect which components were covered in each execution. In addition to the program spectra, information whether that particular spectra corresponds to a failing or a passing execution is also collected in a so-called error vector. In the following, we will refer to the collected information as the (A, e) tuple, where

- $a_{mn} = 1$ if component $1 \leq m \leq M$ was involved in transaction $1 \leq n \leq N$, and 0 otherwise;
- $e_n = 1$ if transaction $1 \leq n \leq N$ failed, and 0 if passed.

[1] The Year 2000 problem is also known as Y2K problem, Y2K bug, or simply Y2K.

	obs			
	c_1	c_2	c_3	e
t_1	1	1	0	1
t_2	0	1	1	1
t_3	1	0	0	1
t_4	1	0	1	0

Fig. 15.5 Hit-spectra matrix

15.4.2 Modus Operandi of Fault Localization

Next, we will illustrate how spectrum-based reasoning to fault localization works using a running example. Consider the hit-spectra matrix in Fig. 15.5 (containing a set of component[2] observations *obs* and transaction outcomes e), with 4 transactions and 3 components.

Spectrum-based reasoning [4, 6, 25] consists on the following:

1. Generate sets of components (candidates) that would explain the observed erroneous behavior.
2. Rank the candidates according to their probability.

Candidate Generation

A diagnostic candidate d is a set of components that are said to be valid if at least one component in d is exercised in all failed transactions. That is,

$$\forall_{n \in 0..N} : e_n = 1 \Rightarrow \exists_{m \in 0..M} : a_{mn} = 1 \wedge d_m \in d$$

We are only interested in minimal candidates,[3] as they can subsume others of higher cardinality. There may be several minimal candidates d_k for a particular spectrum, which constitutes a collection of minimal candidates D.

In our example, the collection of minimal diagnostic candidates that can explain the erroneous behavior are

- $d_1 = \{c_1, c_2\}$
- $d_2 = \{c_1, c_3\}$

Note that the problem of computing the set of minimal candidates is equivalent to computing minimal hitting sets [18].

Candidate Ranking

For each candidate d, the posterior probability is calculated using the naïve Bayes rule[4]

[2] As said in the previous section, by component we mean the unit by which we gather coverage. Basically, components are the columns in the hit-spectra matrix and can represent, e.g., every statement in the source code.

[3] A candidate d is said to be minimal if no valid candidate d' is contained in d.

[4] Probabilities are calculated assuming conditional independence throughout the process.

$$\Pr(d \mid (A, e)) = \Pr(d) \cdot \prod_{n}^{N} \frac{\Pr((A_i, e_i) \mid d)}{\Pr(A_n)}, \tag{15.1}$$

where $\Pr(obs_i)$ is a normalizing term that is identical for all candidates; hence, this term is not considered for ranking purposes.

In order to define $\Pr(d)$, let p_j denote the prior probability that a component c_j is at fault.[5] The prior probability for a candidate d is given by

$$\Pr(d) = \prod_{n \in d} p_n \cdot \prod_{n \notin d} (1 - p_n). \tag{15.2}$$

$\Pr(d)$ estimates the probability that a candidate, without further evidence, is responsible for erroneous behavior—that is, the prior probability of being faulty. It is also used to make to make larger candidates (in terms of cardinality) less probable. In order to bias the prior probability taking observations into account, $\Pr(A_i, e_i \mid d)$ is used. Let g_j (referred to as component goodness) denote the probability that a component c_j performs nominally

$$\Pr((A_i, e_i) \mid d) = \begin{cases} \prod\limits_{j \in (d \cap A_i)} g_j & \text{if } e_n = 0 \\ 1 - \prod\limits_{j \in (d \cap A_i)} g_j & \text{otherwise} \end{cases} \tag{15.3}$$

In cases where values for g_n are not available (which is the case for software components), they can be estimated by maximizing $\Pr((A, e) \mid d)$ (Maximum Likelihood Estimation (MLE) for the naïve Bayes classifier) under parameters $\{g_n \mid n \in d\}$. This MLE-based approach is the BARINEL approach and will be detailed in the next section [6]).

Considering our example, the probabilities for both candidates are

$$\Pr(d_1 \mid (A, e)) = \overbrace{\left(\frac{1}{1000} \cdot \frac{1}{1000} \cdot \left(1 - \frac{1}{1000}\right)\right)}^{\Pr(d)} \times \overbrace{\underbrace{(1 - g_1 \cdot g_2)}_{t_1} \times \underbrace{(1 - g_2)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3} \times \underbrace{g_1}_{t_4}}^{\Pr((A,e)|d)} \tag{15.4}$$

$$\Pr(d_2 \mid A, e) = \overbrace{\left(\frac{1}{1000} \cdot \frac{1}{1000} \cdot \left(1 - \frac{1}{1000}\right)\right)}^{\Pr(d)} \times \overbrace{\underbrace{(1 - g_1)}_{t_1} \times \underbrace{(1 - g_3)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3} \times \underbrace{g_1 \cdot g_3}_{t_4}}^{\Pr((A,e)|d)} \tag{15.5}$$

By performing a MLE for both functions it follows that $\Pr(d_1 \mid (A, e))$ is maximized for $g_1 = 0.47$ and $g_2 = 0.19$, and $\Pr(d_2 \mid A, e)$ is maximized for $g_1 = 0.41$ and $g_3 = 0.50$. Applying the goodness values to both expressions, it follows that

[5] In the context of development-time fault localization, we often approximate p_j as $1/1000$, i.e., 1 fault for each 1000 lines of code.

$\Pr(d_1 \mid (A, e)) = 1.9 \times 10^{-9}$ and $\Pr(d_2 \mid A, e) = 4.0 \times 10^{-10}$, entailing the ranking (d_1, d_2).

15.4.3 The BARINEL *Approach to Compute Goodness*

In the previous section, we have described the *modus operandi* of spectrum-based reasoning to fault localization. A key issue in the outlined approach is to compute the component goodness g_j of each component, as these values influence the posterior probabilities of the diagnostic candidates ($\Pr(d_k)$). The approach was first introduced in [6], as is named BARINEL.

15.4.3.1 Component Goodness Estimation

As mentioned before, the key idea underlying the BARINEL approach is that it computes the g_j for each candidate's faulty components that *maximizes the probability* $\Pr((A, e)|d_k)$ *of the observations* (A, e) *occurring, conditioned on candidate* d_k, yielding statistically perfect estimators for g_j. Following equation (15.3), BARINEL bias the prior probability taking observations into account conditioned on a particular diagnostic candidate d_k using $\Pr(A_i, e_i \mid d_k)$:

$$\Pr((A_i, e_i) \mid d_k) = \begin{cases} \prod\limits_{n \in d_k \wedge a_{mn}=1} g_n & \text{if } e_n = 0 \\ 1 - \prod\limits_{n \in d_k \wedge a_{mn}=1} g_n & \text{if } e_n = 1 \end{cases}$$

Hence, the component goodnesses (g_n) are computed by maximizing $Pr((A_i, e_i)|d_k)$ according to

$$\arg\max_{g_j | j \in d_k} \Pr(e|d_k).$$

Note that this approach implies *optimum* g_n values may differ per diagnostic candidate for the exact same set of components. For instance, suppose a system with $M = 4$ components and the following double- and triple-fault candidates. Let the following be the g_n that optimally explain the observed failures and passes for the double and triple fault $\{0.15, 0.8, 0.4, 1\}$ and $\{0.7, 1, 1, 0.3\}$, respectively. Note that g_n differ for the same component n (e.g., 0.15 vs. 0.7).

The BARINEL approach generalizes over both persistent and intermittent faults. Each component $n \in d_k$ is associated with a component goodness $g_n \in [0, 1]$, which represents a generalization over the classical "normal/abnormal" entries:

- **0** – persistently failing to
- **1** – *healthy*, i.e., faulty but not yielding observed failures.

15.4.3.2 Algorithm

The approach, named BARINEL, is described in detail in Algorithm 15.1 and has three main phases [6]. BARINEL, taking as input (A, e), starts to generate a set of diagnostic candidates $D = \{d_1, \ldots d_k, \ldots, d_{|D|}\}$ using a low-cost, heuristic, and optimized MHS algorithm called STACCATO. The STACCATO algorithm is guided to return a set of limited diagnostic candidates that captures all significant probability mass[6] [2, 19].

Algorithm 15.1: Diagnostic Algorithm: BARINEL (© 2009 IEEE. Reprinted, with permission, from [4]).

Inputs: Activity matrix A, error vector e,
Output: Diagnostic Report D

1: $\gamma \leftarrow \epsilon$
2: $D \leftarrow$ STACCATO$((A, e))$ {Compute MHS}
3: **for** all $d_k \in D$ **do**
4: expr $\leftarrow$ GENERATEPR$((A, e), d_k)$
5: $i \leftarrow 0$
6: $\Pr[d_k]^i \leftarrow 0$
7: **repeat**
8: $i \leftarrow i + 1$
9: **for** all $j \in d_k$ **do**
10: $g_j \leftarrow g_j + \gamma \cdot \nabla \text{expr}(g_j)$
11: $\Pr[d_k]^i \leftarrow$ EVALUATE(expr, $\forall_{j \in d_k} g_j$)
12: **until** $|\Pr[d_k]^{i-1} - \Pr[d_k]^i| \leq \xi$
13: **return** SORT$(D, \Pr)$

In the second phase, $\Pr(d_k|(A, e))$ is computed for each $d_k \in D$ (lines 3 to 14). The function GENERATEPR derives the symbolic formula for $\Pr((A, e)|d_k)$. To illustrate this function, suppose the following observations:

c_1	c_2	e	$\Pr(e_i\|\{1, 2\})$
1	0	1	$1 - g_1$
1	1	1	$1 - g_1 \cdot g_2$
0	1	0	g_2
1	0	0	g_1

According to the BARINEL reasoning, the probability of obtaining (A, e) given $d_k = \{1, 2\}$ equals

$$\Pr((A, e)|d_k) = g_1 \cdot g_2 \cdot (1 - g_1) \cdot (1 - g_1 \cdot g_2)$$

[6]For an efficient implementation of STACCATO, refer to https://github.com/npcardoso/MHS2 or http://mhs2.algorun.org/.

Next, the component goodness values are calculated such that they maximize $\Pr((A, e)|d_k)$. This is calculated by applying a gradient ascent procedure [9] (lines 9–11).

Finally, the diagnoses are ranked according to $\Pr(d_k|(A, e))$, which is computed by EVALUATE according to the posterior Bayes update (line 12):

$$\Pr(d_k|(A, e)) = \frac{\Pr((A, e)|d_k)}{\Pr(A)} \cdot \Pr(d_k)$$

where $\Pr(d_k)$ is the prior probability that d_k is the true fault explanation, $\Pr(A)$ is a normalization factor, and $\Pr((A, e)|d_k)$ is the probability that (A, e) is observed assuming d_k correct.

BARINEL's MLE for single fault diagnostic candidates has been proven to be the intuitive way to estimate the true intermittency parameter (i.e., component goodness values) [6]. Consider the following (A, e) as well as the probability of that occurring (Pr), being g_1 the true intermittency parameter:

c_1	e	$\Pr(e_i\|d_k)$
1	0	g_1
1	0	g_1
1	1	$1 - g_1$

MLE estimates g_1 to be $\frac{2}{3}$. Showing that g_1 maximizes the probability of this particular (A, e) to occur, proves that this is a perfect estimate. As $\Pr(e|\{1\})$ is given by $\Pr(e|\{1\}) = g_1^2 \cdot (1 - g_1)$, the value of g_1 that maximizes $\Pr(e|\{1\})$ is indeed $\frac{2}{3}$. Furthermore, it has also been shown that these estimators yield optimal diagnostic reports, if considering single faults only [6, 42].

There are success stories of applying BARINEL in practice [27, 57] and other areas of research (e.g., [15, 16, 43–45]). This is the case because the time/space complexity of our approach is low—hence, being amenable to large software systems. The complexity is essentially the same as other lightweight approaches modulo a constant factor on the account of the gradient ascent procedure, which exhibits rapid convergence [6]. The approach is available within the GZoltar toolset at http://www.gzoltar.com [17].

15.4.4 Results

To assess the performance improvement of our approach, we generate synthetic observations based on sample (A, e) generated for various values of N, M, and number of injected faults C (cardinality). Essentially, the simulator[7] samples component activity from a Bernoulli distribution with parameter r, i.e., the probability a compo-

[7] Simulator is available at https://github.com/SERG-Delft/sfl-simulator.

nent is involved in a row equals r. For the C faulty components $c_j \in C$ we also set g_j. Thus, the probability of a component being involved *and* generating a failure equals $r \cdot (1 - g)$. A row i in A generates an error ($e_i = 1$) if at least 1 of the C components generates a failure (noisy-or model). Measurements for a specific (N, M, C, r, g) scenario are averaged over 1, 000 sample matrices, yielding a coefficient of variance of approximately 0.02.

We compare the accuracy of our Bayesian framework with the classical framework [35] in terms of a diagnostic performance metric C_d, that denotes the cost of diagnosis (that is, the percentage of statements that a developer needs to inspect before finding finding the actual components at fault) [48]. Given a diagnosis $D = \{d_1, \ldots, d_k, \ldots, d_K\}$, the computation of C_d proceeds as follows: (i) the diagnostic ranking is mapped into a component ranking according to $\Pr(j) = \sum_{k=1}^{K} \Pr(d_k) \cdot h_k[j] / \sum_{k=1}^{K} \Pr(d_k)$, (ii) the ranking is traversed; inspected healthy components contribute to C_d.

For instance, consider a 4-component program with a unique diagnosis $d_1 = \{c_1, c_2, c_4\}$ with an associated $g = \{0.70, 0.20, 1, 0.15\}$, and c_1, c_2 faulty. The first component to be verified/ replaced is the non-faulty c_4, as its goodness is the lowest. Consequently, C_d is increased with $\frac{1}{4}$ to reflect that it was inspected in vain.

Our experiments for the different (N, M, C, r, g) scenarios, lead us to conclude the following:

- For a sufficient large number of executions (N), BARINEL produces optimal diagnosis, being able to correctly pinpoint the true faulty components. One of the reasons for this observation is that the chance that non-faulty components are still within the MHS is low. Furthermore, for single faults ($C = 1$), BARINEL yields optimal diagnosis.
- For small g_j (that is, faulty components are very likely to yield observed failures), C_d converges more quickly than for large g_j as executions involving faulty components are much more likely to fail. For large g_j, BARINEL requires many more observations (larger N) to rank the true faulty components higher.
- More observations (N) are needed to pinpoint true faulty components as the number of faults C increase. The reason is because failure behavior can be caused by much more components, reducing the correlation between failure and a particular component involvement.
- BARINEL is superior to other related approaches for $C \geq 2$. In particular, the other approaches steadily deteriorate for increasing C.

We refrain from detailing the results obtained with BARINEL in many different contexts, instead we have decided to outline the main findings. A detailed description of results—including simulator and real software systems results—can be found in related research papers [3, 4, 6].

15.5 Software Configuration Errors Diagnosis

Software Product Line (SPL) [10, 11, 34] is a new paradigm in the Software Engineering field, which provides the basis for the development of products. This paradigm is based on the identification of a set of core features and their relations in the development of products. SPL methods consist of the process of analyzing related products in order to identify their common and variable features. The main method for the domain analysis of an SPL is based on feature models and Feature-Oriented Domain Analysis [34] represents one of the most used techniques for the domain analysis of feature models.

Feature models (hereinafter FM) describe a model that define features and their relations. FMs enable to reason about certain properties, for instance, the potential number of valid products (set of valid configurations as all valid combination of a selection of features); and whether a particular configuration (selection of features) constitutes a valid product. There are several types of models to design FMs [10]. Although the notation proposed by Czarnecki [24] is the most used in the literature, an example of this notation is shown in the in Fig. 15.6. This notation enables four type of relations between a parent and its child features:

- **Mandatory** relation indicates that a child feature is required as shown in Fig. 15.6 where *computer* feature requires the mandatory sub-feature of *video*, $computer \leftrightarrow video$.
- **Optional** relation indicates that a child feature is optional as shown in Fig. 15.6 where *computer* feature implies an optional sub-feature of *audio*, $computer \rightarrow audio$.
- **Alternative** relation indicates that one of the sub-features must be selected. In general, $a_1, a_2, \ldots, a_n$ alternative sub-features of b, $a_1 \wedge a_2 \wedge \cdots \wedge a_n \leftrightarrow b_{\bigvee_{i<j}(a_i \vee \cdots \vee a_j)}$. In Fig. 15.6 where the *video* feature implies the selection of one *VGA* or *HDMI* feature, $video \leftrightarrow VGA \vee HDMI$.

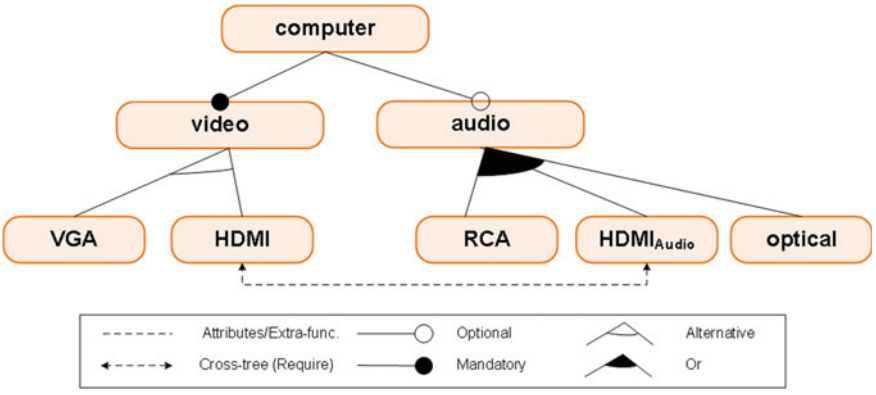

Fig. 15.6 Feature model example

- **Or** relation indicate that at least one of the sub-features must be selected. In general, $a_1, a_2, \ldots, a_n$ sub-features of b, $a_1 \vee a_2 \wedge \cdots \wedge a_n \leftrightarrow b$. In Fig. 15.6, $audio \leftrightarrow (RCA \vee optical \vee HDMI_{Audio}) \wedge \neg(RCA \wedge optical \wedge HDMI_{Audio}))$.

In addition, other relations called cross-tree constraints are allowed. These cross-tree constraints enable to represent constraints that relate features without a direct parent–child relation. The most common are the inclusion and exclusion relations:

- Feature A *requires* feature B, for instance, in Fig. 15.6, $HDMI \leftrightarrow HDMI_{Audio}$.
- Feature A *excludes* feature B, for instance, $\neg$(A $\wedge$ B)).

This graphical notation lacks mechanisms to define certain relations and information of the features. There are some extensions such as proposed by [11, 24] that enable the specification of attributes and extra-functionalities for features. These extensions enable characteristics of features that can be measured to be provided and to include the facility to express relations between these characteristics (extra-functionalities).

In order to determine a software configuration, fault detection or diagnosis feature models can be transformed into formal models. The formal models can be also used to extract information related to the product. This information can vary from: number of configurations (number of all valid configurations), filters (selection of specific characteristics for the features), all valid products (all valid products with certain features), validation (check if a selection of characteristics represent a valid configuration), optimum products (determine the best products according to a criterion), variability (inspect the relation between a set of potential products and certain products), commodity (the relation between certain products and the total of products).

SAT and constraint programming can be used as a formal model to perform feature model analysis since this approach handles integer domains that are used for attributes and optimization functions. Feature models might be transformed into Constraint Satisfaction Problem (CSP), SAT problem, Constraint Optimization Problem (COP), MAX-SAT problem [12, 51] depending on the reasoning question to achieve. Figure 15.7 depicts a transformation example of a feature model to an SAT and MAX-SAT problem.

As aforementioned, other information can be obtained such as the determination whether a configuration is valid or invalid. For a better understanding of these two example configurations related to Fig. 15.6 are given in next as follows:

$$C1 : \{computer, video, HDMI, audio, HDMI_{Audio}\} \tag{15.6}$$

$$C2 : \{computer, video, HDMI, audio, RCA\} \tag{15.7}$$

The first example shows a valid configuration in so far as it has {*computer, video, HDMI,*} $HDMI_{Audio}$ features and the values for C_S, and C_k are valid values. In this case, the configuration is valid since *computer* and *video* are mandatory and the alternative *HDMI* has been selected. Regarding *audio*, $HDMI_{Audio}$ is chosen as requirement of *HDMI* video feature, thus the configuration is valid. Nevertheless, the second

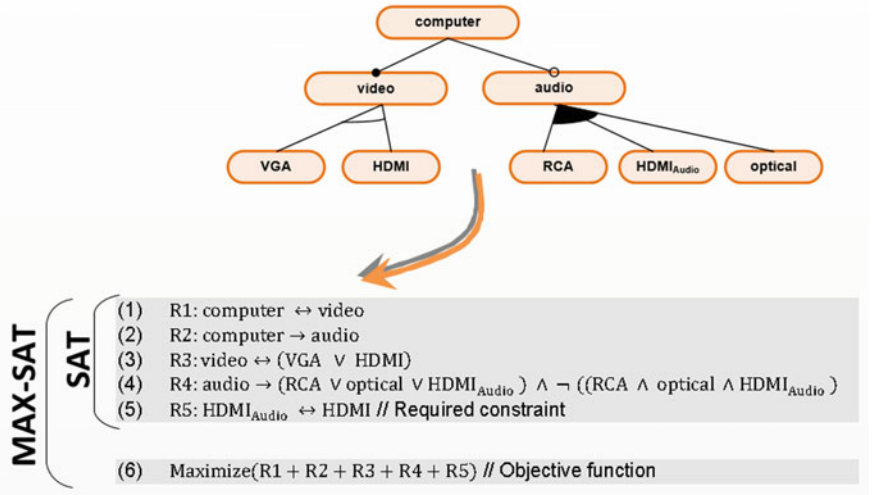

Fig. 15.7 Transformation of a feature model to a SAT

configuration is invalid since the *RCA* feature is in the configuration, however, *HDMI* feature requires $HDMI_{Audio}$ feature to be chosen as a required constraint. This invalid configuration might be diagnosed in order to determine why and what are responsible for this invalid configuration or to determine a faulty feature model composition.

From the point of view of model-based diagnosis, it can be applied in two different ways: (1) diagnose (i.e., detection of faulty software configurations) the feature model in order to identify malformed structures; thus no legal configurations can be computed due to overconstrained structured or bad constructions; (2) given a configuration, diagnose why it is illegal with regard to the feature model. If we assume that feature model is correct and represent a product line, the main problem of diagnosis is in the second case.

Formally, in this context, the System Model *(SM)* is defined as a feature model FM_i and a specific configuration C as an Observational Model *(OM)* which is unsatisfiable. These two components define the System Configuration Error (*SCE*) problem to be diagnosed. The diagnosis strives toward the identification of inconsistencies in the configuration, and specifically the set of features and attributes that produce the inconsistency. This identification might be solved by means of fault diagnosis theory. Following the theory of consistency-based diagnosis proposed in [35] and given in previous chapters, the problem has been formalized as the fault diagnosis of a *SCE* problem in the following definitions.

// Configuration:

(1) computer, video, HDMI, audio, RCA = true

// Reified constraints:

(2) $R1 = computer \leftrightarrow video$
(3) $R2 = computer \rightarrow audio$
(4) $R3 = video \leftrightarrow (VGA \vee HDMI)$
(5) $R4 = audio \rightarrow (RCA \vee optical \vee HDMI_{Audio}) \wedge \neg (RCA \wedge optical \wedge HDMI_{Audio})$
(6) $R5 = HDMI_{Audio} \leftrightarrow HDMI$

// Objective function:

(7) $Maximize(R1 + R2 + R3 + R4 + R5)$

Fig. 15.8 MAX-SAT to diagnose the a problem

Definition 15.4 (*Fault Diagnosis of an SCE*). A fault diagnosis of an *SCE* is a set of features and attributes $\triangle \subseteq C$ such that $\triangle = \{\triangle_{F_j} \cup \triangle_{F_l} \cup \cdots \cup \triangle_{F_m}\}$.

$$FM_i \cup (C - \triangle) \vdash \top \tag{15.8}$$

At this point, the system has been diagnosed by identifying the $\triangle$ as responsible for the inconsistency in the SCE.

In order to do the diagnosis, constraint suspension can be applied. In this case, the problem is translated into a MAX-SAT problem in which *SM* where all the constraints from the original problem are reified, thus is translated to a Boolean constraint; the variables from the configuration, C, are established as a part of the model, and the objective function is established to the satisfaction of the maximum number of the reified constraints. An example of MAX-SAT is shown in Fig. 15.8, it is important to highlight that features that are unestablished in the model are assumed *false* by default.

The MAX-SAT can be solved and it retrieves the information about fault diagnosis as the minimal number of constraints that cannot be satisfied. The solution of this programme is the fault diagnosis, and therefore $\triangle$ as defined previously. In the particulars of the example, the fault diagnosis is $\triangle = R5$ which means that $R5$ cannot be satisfied. The explanation of the fault diagnosis is also obtained for the solver in so far as *HDMI* feature is *true* but $HDMI_{Audio}$ is *false* since it is not within the configuration but *RCA*.

Although a very simple problem has been used as a proof of concept, the real utility of fault detection and diagnosis in software product configuration is demonstrated when the configurations of the system is too complex and these reasoning techniques help to easily and automatically determine the errors made in the configurations.

15.6 Conclusions

In this chapter, we have presented a model-based diagnosis approach to software debugging. We have introduced three different approaches. 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 [5, 33, 38], our approach also contains multiple fault explanations in the diagnostic ranking (typical of model-based approaches [39, 53]).

A constraint model-based diagnosis approach is also proposed. The source code, test cases, and contracts (assertions, precondition, and postcondition) are taken into account in order to obtain a more precise diagnosis. The approach diagnoses defects in contracts and source code. And finally, an approach for diagnosing a feature model in order to identify malformed structures is proposed. Given a configuration, this methodology is able to diagnose why it is illegal with regard to the feature model.

Acknowledgements This work has been partially funded by the Ministry of Science and Technology of Spain (TIN2015-63502-C3-2-R) and the European Regional Development Fund (ERDF/FEDER). This material is based upon work supported by the ERDF's COMPETE 2020 Programme under project No. POCI-01-0145-FEDER-006961 and FCT under project No. UID/EEA/50014/2013.

References

1. Abreu, R., Mayer, W., Stumptner, M., van Gemund, A.J.: Refining spectrum-based fault localization rankings. In: Proceedings of the 2009 ACM symposium on Applied Computing, pp. 409–414. ACM (2009)
2. Abreu, R., Van Gemund, A.J.: A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. SARA **9**, 2–9 (2009)
3. Abreu, R., Zoeteweij, P., Van Gemund, A.J.: An observation-based model for fault localization. In: Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), pp. 64–70. ACM (2008)
4. Abreu, R., Zoeteweij, P., Van Gemund, A.J.: Spectrum-based multiple fault localization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 88–99. IEEE Computer Society (2009)
5. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, pp. 89–98. IEEE Computer Society, Washington, DC, USA (2007)
6. Abreu, R., Zoeteweij, P., Van Gemund, A.J.C.: A new bayesian approach to multiple intermittent fault diagnosis. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI'09, pp. 653–658 (2009)
7. Araki, K., Furukawa, Z., Cheng, J.: A general framework for debugging. IEEE Softw. **8**(3), 14–20 (1991)

8. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. **1**(1), 11–33 (2004)
9. Avriel, M.: Nonlinear Programming: Analysis and Methods. Courier Corporation, North Chelmsford (2003)
10. Batory, D.: Feature models, grammars, and propositional formulas. In: Proceedings of the 9th international conference on Software Product Lines, SPLC'05, pp. 7–20. Springer, Berlin (2005)
11. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Inf. Syst. **35**(6), 615–636 (2010)
12. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated reasoning on feature models. In: LNCS, Advanced Information Systems Engineering: 17th International Conference, CAISE 2005, p. 2005. Springer (2005)
13. Brat, G., Drusinsky, D., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Venet, A., Visser, W., Washington, R.: Experimental evaluation of verification and validation tools on martian rover software. Form. Methods Syst. Des. **25**(2–3), 167–198 (2004)
14. Briand, L.C., Labiche, Y., Sun, H.: Investigating the use of analysis contracts to support fault isolation in object oriented code. In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, 22–24 July 2002, pp. 70–80 (2002)
15. Campos, J., Abreu, R., Fraser, G., d'Amorim, M.: Entropy-based test generation for improved fault localization. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pp. 257–267. IEEE Press (2013)
16. Campos, J., Arcuri, A., Fraser, G., Abreu, R.: Continuous test generation: enhancing continuous integration with automated test generation. In: Proceedings of the 29th ACM/IEEE international conference on Automated Software Engineering, pp. 55–66. ACM (2014)
17. Campos, J., Riboira, A., Perez, A., Abreu, R.: Gzoltar: an eclipse plug-in for testing and debugging. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 378–381. ACM (2012)
18. Cardoso, N., Abreu, R.: A distributed approach to diagnosis candidate generation. In: Portuguese Conference on Artificial Intelligence, pp. 175–186. Springer (2013)
19. Cardoso, N., Abreu, R.: An efficient distributed algorithm for computing minimal hitting sets. In: Proceedings of the 25th International Workshop on Principles of Diagnosis, DX, vol. 14, p. 23 (2014)
20. Ceballos, R., Gasca, R.M., Borrego, D.: Constraint satisfaction techniques for diagnosing errors in design by contract software. ACM SIGSOFT Softw. Eng. Notes **31**(2) (2006)
21. Ceballos, R., Gasca, R.M., Valle, C.D., Borrego, D.: Diagnosing errors in dbc programs using constraint programming. In: Current Topics in Artificial Intelligence, 11th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2005, Santiago de Compostela, Spain, Revised Selected Papers, pp. 200–210 (2005)
22. Ceballos, R., Gasca, R.M., Valle, C.D., Rosa, F.D.L.: A constraint programming approach for software diagnosis. In: Proceedings of the Fifth International Workshop on Automated Debugging, AADEBUG 2003, pp. 187–197 (2003)
23. Ceballos, R., Gasca, R.M., Valle, C.D., Toro, M.: Max-csp approach for software diagnosis. In: Advances in Artificial Intelligence - IBERAMIA 2002, 8th Ibero-American Conference on AI, pp. 172–181 (2002)
24. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. In: Software Process: Improvement and Practice, p. 2005 (2005)
25. De Kleer, J.: Diagnosing multiple persistent and intermittent faults. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI'09, pp. 733–738 (2009)
26. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
27. Gouveia, C., Campos, J., Abreu, R.: Using HTML5 visualizations in software fault localization. In: 2013 First IEEE Working Conference on Software Visualization (VISSOFT), pp. 1–10. IEEE (2013)

28. Hangal, S., Lam, M.S.: Tracking down software bugs using automatic anomaly detection. In: Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, pp. 291–301. IEEE (2002)
29. Hao, D., Zhang, L., Zhang, L., Sun, J., Mei, H.: Vida: Visual interactive debugging. In: IEEE 31st International Conference on Software Engineering, ICSE 2009, pp. 583–586. IEEE (2009)
30. Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L.: An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. STVR J. Softw. Test., Verif., Reliab. **3**, 171–194 (2000)
31. Hofer, B., Riboira, A., Wotawa, F., Abreu, R., Getzner, E.: On the empirical evaluation of fault localization techniques for spreadsheets. In: ICSE '13: Proceedings of the 2013 International Conference on Software Engineering, pp. 68–82. Springer (2013)
32. Jones, C.: The Year 2000 Software Problem: Quantifying the Costs and Assessing the Consequences. ACM Press/Addison-Wesley Publishing Co, Boston (1997)
33. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering, pp. 467–477. IEEE (2002)
34. Kang, K.: Feature-oriented domain analysis (FODA): feasibility study. Technical Report CMU/SEI-90-TR-21 - ESD-90-TR-222, Carnegie Mellon University, Software Engineering Institute (1990)
35. de Kleer, J., Mackworth, A.K., Reiter, R.: Characterizing diagnoses and systems. Artif. Intell. **56**(2–3), 197–222 (1992)
36. Kremenek, T., Ashcraft, K., Yang, J., Engler, D.: Correlation exploitation in error ranking. In: ACM SIGSOFT Software Engineering Notes, vol. 29, pp. 83–93. ACM (2004)
37. Lieberman, H.: The debugging scandal and what to do about it. Commun. ACM **40**(4), 26–30 (1997)
38. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: Sober: Statistical model-based bug localization. In: Proceeding of the ESEC/FSE-13. ACM, Lisbon, Portugal (2005)
39. Mayer, W., Stumptner, M.: Models and tradeoffs in model-based debugging. In: 18th International Workshop on Principles of Diagnosis. Nashville, TN, USA (2007)
40. Mayer, W., Stumptner, M.: Evaluating models for model-based debugging. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 128–137. IEEE Computer Society (2008)
41. Meyer, B.: Applying design by contract. IEEE Comput. **25**(10), 40–51 (1992)
42. Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.: Evaluating and improving fault localization. In: ICSE '17: Proceedings of the 39th International Conference on Software Engineering, pp. 609–620. IEEE Press (2017)
43. Perez, A., Abreu, R., D'Amorim, M.: Prevalence of single-fault fixes and its impact on fault localization. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 12–22. IEEE (2017)
44. Perez, A., Abreu, R., van Deursen, A.: A test-suite diagnosability metric for spectrum-based fault localization approaches. In: Proceedings of the 39th International Conference on Software Engineering, pp. 654–664. IEEE Press (2017)
45. Perez, A., Abreu, R., HASLab, I.T.: Leveraging qualitative reasoning to improve SFL. In: IJCAI, pp. 1935–1941 (2018)
46. Renieres, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003, pp. 30–39. IEEE (2003)
47. Shepard, T., Lamb, M., Kelly, D.: More testing should be taught. Commun. ACM **44**(6), 103–108 (2001)
48. Steimann, F., Frenkel, M., Abreu, R.: Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, pp. 314–324. ACM (2013)
49. Tip, F.: A Survey of Program Slicing Techniques. Centrum voor Wiskunde en Informatica, Amsterdam (1994)

50. Traon, Y.L., Ouabdesselam, F., Robach, C., Baudry, B.: From diagnosis to diagnosability: axiomatization, measurement and application. J. Syst. Softw. **65**(1), 31–50 (2003)
51. Varela-Vaca, Á.J., Gasca, R.M.: Towards the automatic and optimal selection of risk treatments for business processes using a constraint programming approach. Inf. Softw. Technol. **55**(11), 1948–1973 (2013)
52. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Softw. Eng. **42**(8), 707–740 (2016)
53. Wotawa, F., Stumptner, M., Mayer, W.: Model-based debugging or how to diagnose programs automatically. In: Hendtlass, T., Ali, M. (eds.) Developments in Applied Artificial Intelligence, pp. 746–757. Springer, Berlin (2002)
54. Wotawa, F., Weber, J., Nica, M., Ceballos, R.: On the complexity of program debugging using constraints for modeling the program's syntax and semantics. In: Current Topics in Artificial Intelligence, 13th Conference of the Spanish Association for Artificial Intelligence, CAEPIA. Selected Papers, pp. 22–31 (2009)
55. Yang, Q., Li, J.J., Weiss, D.: A survey of coverage based testing tools. In: Proceedings of the 2006 International Workshop on Automation of software test, AST '06, pp. 99–103. ACM, New York, NY, USA (2006)
56. Yilmaz, C., Williams, C.: An automated model-based debugging approach. In: ASE '07: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, pp. 174–183. ACM (2007)
57. Zoeteweij, P., Abreu, R., Golsteijn, R., Van Gemund, A.J.: Diagnosis of embedded software using program spectra. In: 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), pp. 213–220. IEEE (2007)