Diagnosing Errors in DbC Programs Using Constraint Programming

R. Ceballos, R. M. Gasca, C. Del Valle, and D. Borrego

Departamento de Lenguajes y Sistemas Informáticos,
Universidad de Sevilla (Spain)
{ceballos, gasca, carmelo, borrego@lsi.us.es}

Abstract. Model-Based Diagnosis allows to determine why a correctly designed system does not work as it was expected. In this paper, we propose a methodology for software diagnosis which is based on the combination of Design by Contract, Model-Based Diagnosis and Constraint Programming. The contracts are specified by assertions embedded in the source code. These assertions and an abstraction of the source code are transformed into constraints, in order to obtain the model of the system. Afterwards, a goal function is created for detecting which assertions or source code statements are incorrect. The application of this methodology is automatic and is based on Constraint Programming techniques. The originality of this work stems from the transformation of contracts and source code into constraints, in order to determine which assertions and source code statements are not consistent with the specification.

1 Introduction

In recent decades, the Model-Based Diagnosis community has dedicated big efforts to the development of a methodology for the diagnosis of industrial systems based on integrated sensors which are supposed to work correctly. The diagnosis aim is to detect and to isolate the reason of an unexpected behaviour; in other words, to identify the parts which fail in a system. In order to explain a wrong behaviour, the diagnosis process uses a set of observations and a model of the system. In this work, a methodology for diagnosing software is proposed, that is, for isolating errors in programs.

During the recent decades, the number of techniques for automatically testing and debugging software has increased substantially. The testing techniques allow to detect if there are errors in a software development, but it is difficult to isolate the errors if only testing techniques are used. The debugging techniques allow to isolate the errors of a program in an interactive way. Our objective is the application of the Model-Based Diagnosis techniques to a program, in order to isolate the errors of a program.

In order to obtain the errors of a program, we have considered the deep integration between the different areas derived from Software Engineering (Design by Contract and Testing techniques) and Artificial Intelligence (Model-Based Diagnosis and Constraint Programming). Our software diagnosis methodology

has two different steps: first, it is necessary to capture the specification of the correct behaviour of a program (using DbC, testing or expert information); and second, it is necessary to isolate the error (using Model-Based Diagnosis and Constraint Programming techniques).

The main idea is to transform the contracts and source code into an abstract model based on constraints, in a Max-CSP (Maximal Constraint Satisfaction Problem). This model enables the detection of errors in contracts and/or in a source code. A Max-CSP is a framework for modelling and solving real problems as a set of constraints among variables, and a goal function for satisfying the maximum number of constraints.

Design by Contract (DbC) was proposed in [1]. DbC improves the software quality. A previous paper [2] proposed two measures in order to validate the benefits of using DbC: robustness and diagnosability. The robustness is the degree which the software is able to recover from internal faults that would otherwise have provoked a failure. Diagnosability expresses the effort required in the localization of a fault as well as the preciseness allowed by a test strategy on a given system. The results show that robustness and diagnosability improve rapidly with only a few contracts, and for improving the diagnosability, the quantity of the contracts is less important than their quality.

There are different techniques to automate the testing and debugging of a program, such as, slicing techniques [3], model-based debugging[4], model checking [5] or delta debugging [6]. Our proposal is different from these techniques, because we use DbC to obtain a more precise location of the errors in a program. The DbC specification is also used in other techniques for the verification of the component-based programs. The objective of our methodology is not only the verification of the software; our goal is to obtain the isolation of errors in a program, what is a diagnosis methodology. This paper is an improvement and an extension of our own previous work [7].

The remainder of the paper is organized as follows. Section 2 shows the diagnosis framework. Section 3, 4 and 5 explain the framework modules. Finally, conclusions are drawn and future work is outlined.

2 Diagnosis Framework

Figure 1 represents the complete diagnosis process. The process is based on three modules. The Abstract Model Generation (AMG) module receives the source code and contracts of a program and obtains the abstract model. The abstract model is a set of constraints.

The Error Detection (ED) module detects the errors of an executed program. The following definitions specify the kind of errors that this module can detect.

Definition 1. An *infeasible assertion* is a non-viable assertion due to conflicts with previous assertions, fields or variables values. The set of assertions of a contract is verified when a program is executed. An infeasible assertion is a

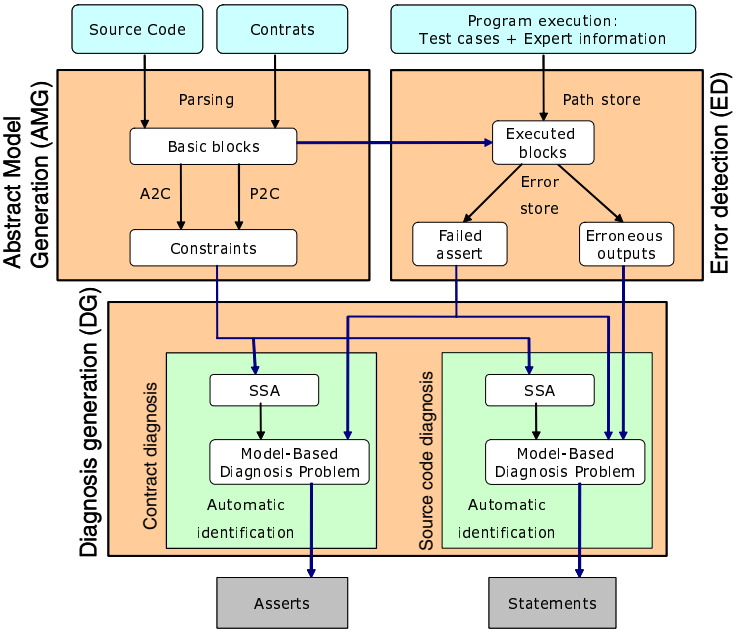

Fig. 1. Diagnosis framework

wrongly designed assertion that cannot be satisfied, and it stops the program execution when it is not necessary.

Definition 2. A *bug* is a statement or a set of statements that does not allow us to obtain the correct result. This paper does not consider the errors detected in compilation time (such as syntax errors), nor dynamic errors (such as exceptions, memory access violations, infinite loops, etc). The bugs under consideration are minor variations of the correct program, such as errors in assignment statements, or errors in the conditions of looping statements or conditional statements.

The ED module receives the set of basic blocks obtained in the AMG module. In our approach, when a program is executed, the information of the executed basic blocks is stored by the ED module. This information will be used for detecting the errors and for diagnosing the system.

The Diagnosis Generation (DG) module obtains the diagnosis for a failed assert or erroneous outputs. The diagnosis of a program will be a set of infeasible assertion and/or bugs. The following subsections describe the modules of the diagnosis framework.

3 Abstract Model Generation (AMG)

In Model-Based diagnosis approaches, a model of the system and a set of observations enable detecting and isolating the reasons of an erroneous behaviour

of a system. The system model simulates the components and their connections by using constraints where the variables represent the inputs and outputs of components.

In an Object-Oriented (OO) program the methods of different objects are linked to obtain an specified behaviour. Each method of an object can be considered as a component which generates a result depending on the input values, the field values, and the statements of the method (which can be considered as subcomponents). The pre-treatment of the source code and program contracts enables to obtain a model of a program. The following subsections show the process for generating this model.

3.1 Determining Basic Blocks

Every OO program is a set of classes. A class is made of methods, fields, assertions, etc. In order to automate the diagnosis of a program, it is necessary to divide the system into subsystems. Each class is transformed into a set of basic blocks (BB) by using a parsing program. These basic blocks can be: blocks of invariants, blocks of static class fields, blocks of object attributes, and blocks of object or class methods.

- Block of invariants (IB): It includes the set of invariants of a class.
- Block of static fields of a class (SB): It includes the set of static field declarations and static code blocks of a class.
- Block of object fields (OB): It includes the set of object field declarations of a class.
- Block of class method or object method (MB): For each constructor or method of a class, a block is obtained. This block is the set of all the statements and assertions (such as preconditions, postconditions or loop invariants) which are included in the method. If the method is static, the block is named a class method block; otherwise it is named an object method block.

Each program class can be transformed into a set of basic blocks (BB_s) equivalent to the $Class_i$. Each method-block is a set of sub-blocks such as conditional blocks (conditional statements) or loop blocks (loops). In our methodology the assignments are defined as the indivisible blocks.

3.2 Program Transformation

The abstract model is a set of constraints which represents the behaviour of the contracts (assertions) and the source code (statements) of a program. Our approach uses the function *A2C* (assertion to constraints) for transforming an assertion into constraints. The transformation of the source code into constraints appeared in a previous work [7]. The process is based on the transformation of statements of each basic block. Our approach uses the function *P2C* (program to constraints) for transforming a source code into constraints. The main ideas of the transformation are:

– Indivisible blocks:
  Assignments: {*Ident := Exp*}
  The assignment statement is transformed into the following equality constraint: {Ident = Exp}. If the assignment statement is not a part of the minimal diagnosis, then the equality between the assigned variable and the assigned expression must be satisfied.
  Method calls and return statements: For each method call, the constraints defined in the precondition and postcondition of the method are added. The method calls allow to link basic blocks and to obtain the order of the blocks in the program execution.
– Conditional blocks: {*if (cond)* {If_{Block}} *else* {$Else_{Block}$}}
  There are two possible paths in a conditional statement depending on the inputs of the condition. The constraints of a conditional statement include the condition and the inner statements of the selected path (only one of the two possible paths is executed).
– Loop blocks: {*while (cond)* {$Block_{Loop}$}}
  In a loop, the number of iterations depends on the inputs. Each iteration is transformed into a conditional statement. The structure of a loop is simulated as a set of nested conditional statements. In [7] a method is proposed to reduce the model to less than n iterations, but if the invariant of the loop exists, the diagnosis process is more precise.

4 Error Detection (ED)

This module detects the errors of an execution of a program. This module can detect failed assertions or erroneous outputs. A failed assertion is an assertion which stops the program due to the conflicts with previous assertions, fields or variables values. The erroneous outputs are detected when outputs are different of the correct results. These correct results must be specified by contracts, test cases or expert information. The Error Detection (ED) module receives the set of basic blocks obtained by the AMG module. At this point, it is important to introduce one concept from the imperative programming.

Definition 3. A *Path* is the sequence of statements that are executed. The executed path of a program depends on the inputs. Conditional statements, loop statements and method calls enable the incorporation of more or less statements to a program execution.

In a Object-Oriented Imperative Language, a path of a program is a sequence of basic blocks obtained by linking constructors and method calls. In our approach, the diagnosis of the program is based on a model generated by linking the constraints obtained from the basic blocks of the executed path.

The erroneous outputs are detected by using DbC specification, test cases and expert information. If the DbC is too weak, an expert can decide which should be the correct inputs and outputs. Testing techniques enable the selection of the

most significant observations in order to detect errors in programs. In [8], the objectives and complications of good Testing are set out.

Definition 4. A *Test case* (TC) is a set of inputs (class fields, parameters or variables), execution preconditions, and expected outcomes, which are developed for a particular objective, such as to exercise a particular program path or to verify the compliance with a specific requirement.

When a program is executed by using a test case, the information about the executed basic blocks is stored, such as the executed path, values of the variables, and the results of the assertion evaluations. This information is necessary for the following Diagnosis Generation module.

5 Diagnosis Generation (DG)

The DG module receives the errors detected by the ED module. These errors can be failed assertions or erroneous outputs. If the error is a failed assertion, the cause of the errors can be infeasible assertions and/or bugs. If the error is an erroneous output, the cause of the problem must be due to bugs. The following subsection shows the transformation of an abstract model into a diagnosis problem in order to detect infeasible assertions and/or bugs.

5.1 Diagnosis Problem

A diagnosis is a hypothesis about which changes are necessary in a program to obtain a correct behavior. The definition of diagnosis, in Model Based Diagnosis (MDB), is built up from the notion of the abnormal predicate [9]: $AB(c)$ is a boolean variable which holds when a component c of the system is abnormal. For example, an adder component is abnormal if the output of the adder is not the sum of its inputs. A diagnosis specifies whether each component of a system is abnormal or not. In order to clarify the diagnosis process, some definitions must be established.

Definition 5. *System model* (SM) is a tuple {PD(PC), TC} where: PC are the program components, that is, the finite set of statements and asserts of a program; PD is the program (statements and asserts) description, that is, the set of constraints obtained of the PC; and TC is a test case.

Definition 6. *Diagnosis*: Let $\mathcal{D} \subseteq$ PC, $\mathcal{D}$ is a diagnosis if PD' $\cup$ TC is satisfiable, where PD' = PD(PC $-$ $\mathcal{D}$).

Definition 7. *Minimal Diagnosis* is a diagnosis $\mathcal{D}$ that for no proper subset $\mathcal{D}$' of $\mathcal{D}$, $\mathcal{D}$' is a diagnosis. The minimal diagnosis implies to modify the smallest number of program statements or assertions.

The goal is to identify the minimal diagnosis consistent with a test case. The constraints of the PD of a program will be obtained as it was explained in

Table 1. Program model of the modified Toy problem

TC	PC	PD
Inputs : {a = 3, b = 2, c = 2, d = 3, e= 3}	S1 : int x = a * c	(AB(S1) ∨ (x == a * c))∧
	S2 : int y = b * d	(AB(S2) ∨ (y == b * d))∧
Outputs : {f = 12, g = 12}	S3 : int z = c + e	(AB(S3) ∨ (z == c + e))∧
Test Code: S1 .. S5	S4 : int f = x + y	(AB(S4) ∨ (f == x + y))∧
	S5 : int g = y + z	(AB(S5) ∨ (g == y + z))∧
	Post : f = a * c + b * d ∧ g = b * d + c * e	(f == a * c + b * d)∧ (g == b * d + c * e)

section 3. When a program is executed, the order of the assertions and statements is very important. It is necessary to maintain this order in the abstract model (based on constraints). Hence, the program under analysis is transformed into a static single assignment (SSA) form. In SSA form, only one assignment is made to each variable in the whole program. For example, the code *x=a*c; ...x=x+3;...* {*Post:x =...*} is changed to *x1=a*c; ...x2=x1+3;...* {*Post:x2 =...*}. More details about SSA form are shown in [10].

Table 1 shows the PD of the toy program derived from the standard toy problem used in the diagnosis community [9]. The program does not reach the correct output because the third statement is an adder instead of a multiplier. In order to obtain the minimal diagnosis a Maximal Constraint Satisfaction Problem (Max-CSP) is generated. A Max-CSP is a CSP with a goal function to maximize. The objective is to find an assignment of the AB variables that satisfies the maximum number of the PD constraints: Goal Function = Max(N AB(i) : AB(i) = *false*). The diagnosis process by using a Max-CSP was shown in a previous work [11]. For example, by using a Max-CSP, the minimal diagnoses for the toy program would be: {{S3}, {S5}, {S1, S2}, {S2, S4}}.

5.2 Diagnosing Contracts

The assertions can be checked by using test cases or not.

- **Diagnosis of assertions without using test cases:** Two kinds of checks are proposed at this point:
  - Checking the invariants of a class: The invariants must always be satisfied. Hence, a Max-CSP is generated by using the invariants of each class in order to check if all the invariants of a class can be satisfied together.
  - Checking the assertions of the methods: The precondition and postcondition of a method must be feasible with the invariants of a class. In order to detect conflicts between the precondition or the postcondition with the invariants, a Max-CSP is generated by using the constraints associated with the assertions.

  The solutions of these Max-CSP problems enable the verification of the feasibility of assertions.

```
/**
 *  @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. 2. Interface *Account* and class *AccountImp* source code

Table 2. Diagnosis of the method *Withdraw* by using a test case

TC	Inputs:	{balance@pre = 0, withdrawal = 100}
	Outputs:	{balance = 0}
	Test code:	Method Withdraw
PD	Inv.	(AB(Inv) $\vee$ (balance@pre >= 0)) $\wedge$
	Pre.	(AB(Pre) $\vee$ (withdrawal > 0)) $\wedge$
	Post.	(AB(Post) $\vee$ (balance = balance@pre - withdrawal)) $\wedge$
	Inv.	(AB(Inv) $\vee$ (balance >= 0))

– **Diagnosis of assertions by using test cases:** It is possible to obtain more information about the viability of the method assertions by applying test cases to the sequence {invariants + precondition + postcondition + invariants } in each method.

Example. In order to clarify the methodology, the class *AccountImp* is used. This class implements the interface *Account* that simulates a bank account. It is possible to deposit money and to withdraw money. Figure 2 shows the source code. The method *deposit* has a bug, because it *decreases* the account balance.

Table 2 shows the PD for the method *withdraw*. In this example there are four constraints: the invariants before and after the method, the precondition, and the postcondition. The test case specifies that the initial and final balance of the account must be 0, when a positive amount is withdrawn.

The invariant specifies that the balance must be equal or greater than zero when the method finishes, but if this invariant is satisfied, it implies that the precondition and the postcondition could not be satisfied together. The postcondition implies that *balance* = *balance@pre* - *withdrawal*, that is, $0-withdrawal > 0$, and this is impossible if the *withdrawal* is positive. The error stays in the precondition, since this precondition is not strong enough to stop the program execution when the withdrawal is not equal or greater than the balance.

5.3 Diagnosing Source Code (with Assertions)

As we are looking for the minimal diagnosis, this work proposes to use a Max-CSP in order to maximize the number of satisfied constraints of the PD. The constraint obtained by the assertions must be satisfied, because these constraints give us information about the correct behaviour. But the constraints obtained from the source code can be satisfied or not. The result of the diagnosis process depends on the outputs obtained by using the test case and the final state of the program:

- State 1: If the program ended up with a failed assertion, and did not reach the end as specified in the test case, then the error can be a strict assertion (the assertion is very restrictive) or one or more bugs before the assertion. In order to determine the cause of the problem, the program should be executed again without the assertion, in order to check if the program can finish without the assertion.
- State 2: If the program ends, but the result is not the one specified by the test case, then the error can be a bug, or an assertion which is not enough restrictive (this enables executing statements which obtain an incorrect result). If the error is a bug, the resolution of the Max-CSP provides the minimal diagnosis that includes the bug. If the error is due to a weak assertion, then a deeper study of the assertions is necessary. At present, we are researching this point.

Example. In order to clarify the methodology, an example is proposed in Table 3. This is an account with an initial balance of 300 units. Two sequential operations are applied (a withdrawal of 300 units and a deposit of the same quantity). The final balance should be 300 units. The constraints solver determines that the error is caused by the statement included in the method *deposit*. If the method is examined closely, it can be seen that there is a subtraction instead of an addition. The postcondition of this method was too weak, and did

Table 3. Class *AccountImp* checking by using a test case

TC	Inputs:	{balance@pre = 300, withdrawal = 300, income = 300}
	Outputs:	{balance = 300}
	Test code:	S1: account.withdraw(withdrawal) S2: account.deposit(income)
PD	Inv.	balance0 >= 0 ∧
	Pre.	withdrawal > 0 ∧
	Code	(AB(S1) ∨ (balance1 = balance0 - withdrawal)) ∧
	Post.	balance1 = balance0 - withdrawal ∧
	Inv.	balance1 >= 0 ∧
	Pre.	income > 0 ∧
	Code	(AB(S2) ∨ (balance2 = balance1 - income)) ∧
	Post.	balance2 >= 0 ∧
	Inv.	balance2 >= 0

not permit to detect the error. The statement included in the method *withdraw* influences the final result of the balance; however, it is not a possible bug because of the postcondition, since it is strong enough to guarantee the balance value at the end of the method. We can conclude that if the contracts are strong enough the diagnosis will be better.

6 Conclusion and Future Work

This paper is an improvement of a previous work [7] in order to automate the diagnosis of DbC software. This approach incorporates a more precise way to diagnose software since more DbC characteristics are incorporated. As shown by the studied examples, the stronger the contracts are, the better the diagnosis is, because the framework has more information of the expected behaviour.

A more complex diagnosis process is being developed in order to obtain a more precise minimal diagnosis. We are extending this methodology to include all the characteristics of an Object-Oriented language, such as inheritance, exceptions and concurrence. Another important direction is the application of the methodology to more complex and real examples where the transformation of the system into constraints is less straightforward.

Acknowledgements

This work has been funded by the Spanish Ministry of Science and Technology (DPI2003-07146-C02-01) and the European Regional Development Fund (ERDF/ FEDER).

References

1. Meyer, B.: Applying design by contract. IEEE Computer **25** (1992) 40–51
2. Traon, Y.L., Ouabdesselam, F., Robach, C., Baudry, B.: From diagnosis to diagnosability: Axiomatization, measurement and application. The Journal of Systems and Software **1** (2003) 31–50
3. Weiser, M.: Program slicing. IEEE Transactions on Software Engineering **4** (1984) 352–357
4. Mateis, C., Stumptner, M., Wieland, D., Wotawa., F.: Model-based debugging of java programs. In: AADEBUG. (2000)
5. Groce, A., Visser, W.: Model checking java programs using structural heuristics. In: ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, Roma, Italy (2002) 12–21
6. Cleve, H., Zeller, A.: Locating causes of program failures. In: 27th International Conference on Software Engineering (ICSE 2005), St. Louis, Missouri (2005)
7. Ceballos, R., Gasca, R.M., Valle, C.D., Rosa, F.D.L.: A constraint programming approach for software diagnosis. In: AADEBUG Fith International Symposium on Automated and Analysis-Driven Debugging, Ghent, Belgium (2003) 187–196
8. Binder, R.: Testing Object-Oriented Systems : Models, Patterms, and Tools. Object Technology Series. Addison Wesley (2000)

9. de Kleer, J., Mackworth, A., Reiter, R.: Characterizing diagnoses and systems. Artificial Intelligence **2-3** (1992) 197–222
10. Alpern, B., Wegman, M., Zadeck, F.K.: Detecting equality of variables in programs. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Diego, California (1988) 1–11
11. Ceballos, R., del Valle, C., Gómez-López, M.T., Gasca, R.M.: CSP aplicados a la diagnosis basada en modelos. Revista Iberoamericana de Inteligencia Artificial **20** (2003) 137–150