Constraint satisfaction techniques for diagnosing errors in Design by Contract software[*]

Rafael Ceballos
Dpto. Lenguajes y Sistemas Informáticos, U. de Sevilla
Avda. Reina Mercedes s/n, CP 41012
Seville, Spain

Rafael Martínez Gasca
Dpto. Lenguajes y Sistemas Informáticos, U. de Sevilla
Avda. Reina Mercedes s/n, CP 41012
Seville, Spain

Diana Borrego
Dpto. Lenguajes y Sistemas Informáticos, U. de Sevilla
Avda. Reina Mercedes s/n, CP 41012
Seville, Spain

ABSTRACT

Design by Contract enables the development of more reliable and robust software applications. In this paper, a methodology that diagnoses errors in software is proposed. This is based on the combination of Design by Contract, Model-based Diagnosis and Constraint Programming. Contracts are specified by using assertions. These assertions together with an abstraction of the source code are transformed into constraints. The methodology detects if the contracts are consistent, and if there are incompatibilities between contracts and source code. The process is automatic and is based on constraint programming.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Assertion checkers, Correctness proofs, Programming by contract; D.2.5 [**Testing and Debugging**]: Diagnostics; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Assertions Invariants Pre- and post-conditions

General Terms

Verification, Design by Contract, Diagnosis

1. INTRODUCTION

Design by Contract was proposed in [5]. This work specified that the major component of quality in software is the ability to perform its job according to the specification. The software quality is especially important in Object-Oriented (OO) methodology because of the software reusability. In a recent work, [1] it is showed that contracts are useful for fault isolation. By using contracts, the fault isolation and diagnosability is significantly improved.

In this work, a methodology for diagnosing software is proposed, that is, for detecting and locating faults in programs. The main idea is the transformation of the contracts and source code into an abstract model based on constraints. The methodology detects if the contracts are consistent, and if there are incompatibilities between contracts and source code.

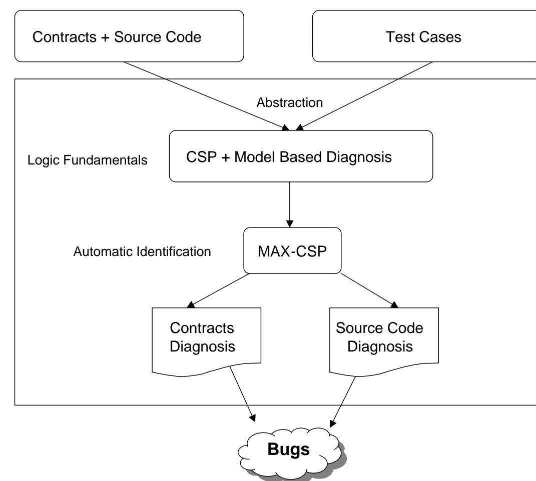

Figure 1: Diagnosis framework

2. DIAGNOSIS FRAMEWORK

Figure 1 shows the completed diagnosis process. The process obtains an abstract model based on the source code, contracts and test cases. The diagnosis of a program is a set of infeasible assertion and/or erroneous statements. These definitions specify the kind of errors that can be detected.

Definition 1. An *infeasible assertion* is a non-viable assertion due to conflicts with previous assertions or statements. The set of assertions of a contract are verified when a program is executed. An infeasible assertion is a wrongly designed assertion that cannot be satisfied and it stops the program execution when it is not specified.

Definition 2. An *erroneous statement* is a statement or a set of statements that are faults since they do not allow the

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

```
/**
 *  @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);

    /**
     *  @pre interest >= 0
     *  @post getInterest() == interest
     */
    public void setInterest (double interest);

    public double getInterest ();

    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;
    }

    public double getInterest() {
       return this.interest;
    }

    public void setInterest(double interest) {
       this.interest = interest;
    }

}
```

Figure 2: Interface *Account* and class *AccountImp* source code

correct results to be obtained. The errors under consideration are minor variations of the correct program, such as errors in loop conditions or errors in assignment statements. This paper does not consider errors detected in compilation time (such as syntax errors), nor dynamic errors (such as exceptions, memory access violations, infinite loops, etc).

The following sections explain the phases of the diagnosis process.

3. ABSTRACT MODEL GENERATION

In model-based diagnosis approaches, a model of the system components enables detecting, identifying and isolating the reason of an unexpected behaviour of a system. In a Object-Oriented program the methods of different objects are linked in order to obtain the specified behaviour. Each method of a object can be considered as a component, which generates a specified result. The pretreatment of the source code and program contracts enables obtaining an abstract model of a program. This abstract model allows to diagnose errors in programs. The following subsections shows the process for generating abstract model.

3.1 Determining basic blocks

Every OO program is a set of classes. In order to automate the diagnosis of a program it is necessary to divide the system into subsystems. Each program classes is transformed into a set of basic blocks. These basic blocks (BB) can be: blocks of invariants (IB), blocks of static class fields (SB), blocks of object attributes (OB), and blocks of object or class methods (MB). A IB includes the set of invariants of a class. A SB includes the set of static field declarations and static code blocks of a class, and a OB includes the set of object field declarations of a class. A MB is the set of all the statements and assertions (such as preconditions, postconditions or loop invariants) included in the method.

Each program class can be transformed into a set of basic blocks (BB_s) equivalent to the $Class_i$. When a program is executed, the microprocessor links the basic blocks. The order of these blocks can be represented as a *Control Flow Graph* (CFG). The CFG is a directed graph that represents the control structure of a program. A CFG is a set of sequential blocks and decision statements. A *Path* is the sequence of statements of the CFG that is executed. The following sections will use the basic blocks of an executed path in order to obtain the constrains of the abstract model.

3.2 SSA form

The order of the constraints is not important for solving a CSP. But 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. The program under analysis is translated into a static single assignment (SSA) form. This step maintains the execution sequence when the program is translated into constraints. 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 =...*}.

3.3 Program transformation

The abstract model is a set of constraints which simulate 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 to constraints appears in previous work [3]. The process is based on the transformation of each statement of the path. The main ideas of the transformation are:

- Indivisible blocks:

  Assignments: {*Ident* := *Exp*}
  The assignment statement is transformed into the fol-

Table 1: The modified toy problem model

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

lowing 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. If we find a recursive method call, this internal method call is supposed to be correct in order to obtain the formal verification of the recursive calls.

- 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 cycles depends on the inputs. Each cycle is transformed into a conditional statement. The structure of a loop is simulated as a set of nested conditional statements. If the invariant of the loop exists, the diagnosis process is more precise.

3.4 Test cases

Testing techniques enables the selection of those observations which are the most significant for detecting bugs in a program. 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 compliance with a specific requirement. The values of a test case must satisfy the DbC specification. In our approach when a program is executed by using a test case, the information of the executed basic blocks is stored. This information is necessary for the diagnosis of the system, since it contains which are the statements of the executed path.

4. DIAGNOSIS PROBLEM

A diagnosis is a hypothesis for how a program must change in order to obtain a correct behavior. The definition of diagnosis, in Model Based Diagnosis (MDB), is built up from the notion of abnormal [4]: AB(c) is a boolean variable which holds when a component c of the system is abnormal. For example, an adder component is not abnormal if the output of the adder is 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 3. *System model*(SM) is a tuple {PC, PD, TC} where: PC are the program components, that is, the finite set of statements and asserts of a program; PD is the program description, that is, the set of constraints (abstract model, AM) obtained of the PC, PD = AM(PC); and TC is a test case.

Definition 4. *Diagnosis*: Let $\mathcal{D} \subseteq$ PC, $\mathcal{D}$ is a diagnosis if PD' $\cup$ TC is satisfiable, where PD' = PD(PC $-$ $\mathcal{D}$).

The goal of diagnosis is to identify, and refine, the set of diagnoses consistent with the test case.

Definition 5. *Minimal Diagnosis* is a diagnosis $\mathcal{D}$ that for no proper subset $\mathcal{D}$' of $\mathcal{D}$ is $\mathcal{D}$' a diagnosis. The minimal diagnoses imply to modify the smallest number of program statements or assertions.

The rules described in the section 3 enables implementing the function AM(Abstract Model) which generates the PD of a program. Table 1 shows the PD of the toy problem program, this is derived from the standard toy problem used in the diagnosis community [4]. The program can not reach the correct output because the third statement is a adder instead of a multiplier.

In order to obtain the minimal diagnosis it is generated a Maximal Constraint Satisfaction Problem (Max-CSP). A Max-CSP is a CSP with a goal function. The objective is to find an assignment of the AB variables that satisfies the maximum number of the PD constraints: Goal Function = Max(N i : AB(i) = *false*). A constraints solver will generate the different solutions of the Max-CSP. The diagnosis process by using a Max-CSP is shown previous work [2]. For example, by using a Max-CSP, the minimal diagnoses in the toy program will be: {{S3}, {S5}, {S1, S2}, {S2, S4}}.

5. DIAGNOSING PROGRAMS

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 and contracts. The method *deposit* has a bug, in that it *decreases* the account balance. In the first phase, assertions are checked in two different ways: without test cases and with test cases. In the second phase the source code with assertions is checked by using test cases.

5.1 Diagnosis of assertions without test cases

Two kinds of checks are proposed at this point: 1) a Max-CSP with all the invariants of each class, in order to check if all the invariants of a class can be satisfied together; and 2) a Max-CSP with the assertions of the methods and the invariants, in order to check if the precondition and postcondition of a method is feasible with the invariants of a class. The solutions of these Max-CSP problems enable the verification of the feasibility of assertions.

5.2 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.

Table 2: Diagnosis of the method *Withdraw*

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

Example. Table 2 shows the PD for the method *withdraw* verification. The test case specified that the initial balance must be 0 units and, when a positive amount is withdrawn, the balance must preserve the value 0. 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 problem resides 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 of the account.

5.3 Diagnosis of source code and assertions

The diagnosis of a program is the set of those statements which include the errors. We are looking for the minimal diagnosis, that is, it is necessary, by using a Max-CSP, 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. The diagnosis process result depends on the the final situation of the program:

Situation 1: If the program ended up with a failed assertion, and did not reach the end as specified in the test case, the problem can be a strict assertion (the assertion is very restrictive) or one or more erroneous statements before the assertion. In order to determine the cause of the problem, the program should be executed again without the assertion, in order to deduce if the program can finish without the assertion. If this happens, the assertion is very strict. If the program does not finish, the problem is due to the code up to the point of the assertion.

Situation 2: If the program ends, but the result is not the one specified by the test case, then the problem can be a wrong statement, or an assertion which is not sufficiently restrictive (this enables executing statements which obtain an incorrect result). If the problem is a wrong statement, the resolution of the Max-CSP problem provides the minimal diagnosis that includes the bugs. If the problem is due to a weak assertion then a deeper study of the assertions is necessary.

Example. Table 3 shows an account with an initial balance of 300 units. Two sequential operations are applied: a withdrawal of capital of 300 units, and a deposit of the same quantity. The constraint 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 is too weak, and did not detect this problem.

Table 3: Diagnosis of the class *AccountImp*

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
	Inv.	balance1 >= 0
	Pre.	income > 0
	Code	(AB(S2) ∨ (balance2 = balance1 - income))
	Post.	balance2 >= 0
	Inv.	balance2 >= 0

6. CONCLUSION AND FUTURE WORK

In order to automate the diagnosis of software with contracts, the combination of techniques from different subjects is proposed, such as Constraint Programming, Model-Based Diagnosis, and Design by Contract. This paper is an improvement of previous work [3], and incorporates a more precise way to diagnose software since more characteristics of Design by Contract are incorporated. The methodology detects if the contracts are consistent, and if there are incompatibilities between contracts and source code. A more complex diagnosis process is being developed in order to obtain a more precise minimal diagnosis. We are extending the methodology to include all the characteristics of an Object-Oriented language, such as inheritance, exceptions and concurrence.

7. REFERENCES

[1] L. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object-oriented code. In *International Symposium on Software Testing and Analysis*, Roma, Italy, 2002.

[2] R. Ceballos, C. del Valle, M. T. Gómez-López, and R. M. Gasca. CSP aplicados a la diagnosis basada en modelos. *Revista Iberoamericana de Inteligencia Artificial*, 20:137–150, 2003.

[3] R. Ceballos, R. M. Gasca, C. D. Valle, and F. D. L. Rosa. A constraint programming approach for software diagnosis. In *AADEBUG*, pages 187–196, Ghent, Belgium, September 2003.

[4] J. de Kleer, A. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 2-3(56):197–222, 1992.

[5] B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.