Max-CSP Approach for Software Diagnosis

R. Ceballos, Rafael M. Gasca, Carmelo Del Valle, and Miguel Toro

Languages and Computer Systems Department, University of Seville
Computer Engineering Superior Technical School,
Avenida Reina Mercedes s/n 41012 Sevilla(Spain)

Abstract. In software development is essential to have tools for the software diagnosis to help the programmers and development engineers to locate the bugs. In this paper, we propose a new approach that identifies the possible bugs and detect why the program does not satisfy the specified result. A typical diagnosis problem is built starting from the structure and semantics of the original source code and the precondition and postcondition formal specifications. When we apply a determined test case to a program and this program fails, then we can use our methodology in order to obtain automatically the sentence or the set of sentences that contains the bug. The originality of our methodology is due to the use of a constraint-based model for software and Max-CSP techniques to obtain the minimal diagnosis and to avoid explicitly to build the functional dependency graph.

1 Introduction

Software diagnosis allows us to identify the parts of the program that fail. Most of the approaches appeared in the last decade have based the diagnosis method on the use of models (Model Based Diagnosis). The JADE Project investigated the software diagnosis using Model Based Debugging. The papers related to this project use a dependence model based on the source code. The model represents the sentences and expressions as if they were components, and the variables as if they were connections. They transform Java^{TM} constructs into components. The assignments, conditions, loops, etc. have their corresponding method of transformation. For a bigger concretion the reader can consult [10][11].

Previously to these works, it has been suggested the *Slicing* technique in software diagnosis. This technique identifies the constructs of the source code that can influence in the value of a variable in a given point of the program [12][13]. *Dicing*[9] is an extension to this technique. It was proposed as a fault localization method for reducing the number of statements that need to be examined to find faults with respect to *Slicing*. In the last years, new methods [3][5] have arisen to automate software diagnosis process.

In this work, we present an alternative approach to the previous works. The main idea is to transform the source code into constraints what avoids the explicit construction of the functional dependencies graph of the program variables. The following resources must be available to apply this methodology: Source code,

precondition and postcondition. If the source code is executed in some of the states defined by the precondition, then it is guaranteed that the source code will finish in some of the states defined by the postcondition. Nothing is guaranteed if the source code is executed in an initial state that broke the precondition.

We use Max-CSP techniques to carry out the minimal diagnosis. A Constraint Satisfaction is a framework for modelling and solving real-problems as a set of constraints among variables. A Constraint Satisfaction is defined by a set of variables X={X_1,X_2...,X_n} associated with a domain, D={D_1,D_2,...,D_n} (where every element of D_i is represented by set of v_i), and a set of constraints C={C_1,C_2,...,C_m}. Each constraint C_i is a pair (W_i,R_i), where R_i is a relation $R_i \subseteq D_{i1}$x...xD_{ik} defined in a subset of variables $W_i \subseteq X$.

If we have a CSP, the Max-CSP aim is to find an assignment that satisfies most constraints, and minimize the number of violated constraints. The diagnosis aim is to find what constraints are not satisfied. The solutions searched with Max-CSP techniques is very complex. Some investigations have tried to improve the efficiency of this problem,[4][8].

To carry out the diagnosis we must use Testing techniques to select which observations are the most significant, and which give us more information. In [1] appears the objectives and the complications that a good Testing implies. It is necessary to be aware of the Testing limits. The combinations of inputs and outputs of the programs (even of the most trivial) are too wide.

The programs that are in the scope of this paper are:

- Those which can be compiled to be debugged but they do not verify the specification Pre/Post.
- Those which are a slight variant of the correct program, although they are wrong.
- Those where all the appeared methods include precondition and postcondition formal specification.

This work is part of a global project that will allow us to perform object oriented software diagnosis. This project is in evolution and there are points which we are still investigating.

The work is structured as follows. First we present the necessary definitions to explain the methodology. Then we indicate the diagnosis methodology: obtaining the PCM and the minimal diagnosis. We will conclude indicating the results obtained in several different examples, conclusions and future work in this investigation line.

2 Notation and Definitions

Definition 1. Test Case(TC): It is a tuple that assigns values to the observable variables. We can use Testing techniques to find those *TCs* that can report us a more precise diagnosis. The Testing will give us the values of the input parameters and some or all the outputs that the source code generates. The inputs that the Testing provides must satisfy the precondition, and the outputs must satisfy

the postcondition. The Testing can also provide us an output value which cannot be guaranteed by the postcondition. If this happens, an expert must guarantee that they are the correct values. Therefore, the values obtained by the Testing will be the correct values, and not those that we can obtain by the source code execution. We will use test cases obtained by white box techniques. In example 1a (see figure 3) a test case could be: $TC \equiv \{a=2,b=2,c=3,d=3,e=2,f=12,g=12\ \}$

Definition 2. Diagnosis Unit Specification: It is a tuple that contemplates the following elements: The Source Code (SC) that satisfies a grammar, the precondition assertion (Pre) and the postcondition assertion (Post). We will apply the proposed methodology to this diagnosis unit using a TC and then, we will obtain the sentence or set of sentences that could be possible bugs.

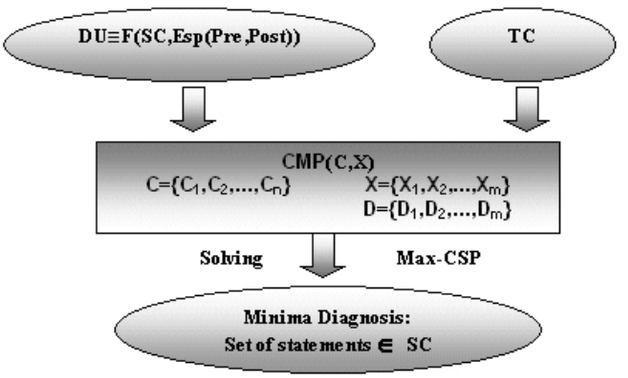

Fig. 1. Diagnosis Process

Definition 3. Observable Variables and Non Observable Variables: The set of observable variables (*Vobs*) will include the input parameters and those output variables whose correct value can be deduced by the TC. The rest of the variables will be non observable variables (*Vnobs*).

Definition 4. Program Constraint-Based Model (PCM): It will be compound of a constraints network C and a set of variables with a domain. The set C will determine the behavior of the program by means of the relationships among the variables. The set of variables set will include (*Vobs*) and (*Vnobs*). Therefore: *PCM(C,Vobs,Vnobs)*

3 Diagnosis Methodology

The diagnosis methodology will be a process to transform a program into a Max-CSP; as it appears in figure 1. The diagnosis process consists of the following steps:

1. Obtaining the PCM:
   - Determining the variables and their domains.
   - Determining the PCM constraints.
2. Obtaining the minimal diagnosis:
   - Determining the function to maximize.
   - Max-CSP resolution.

3.1 Obtaining the PCM

Determining the variables and their domain. The set of variables X={X_1, X_2... ,X_n} (associated to a domain D={D_1,D_2,... ,D_n}) will be compound of *Vobs* and *Vnobs*. The domain or concrete values of each variable will be determined by the variable declaration. The domain of every variable will be the same as the compiler fixes for the different data types defined in the language.

Determining the PCM constraints. The PCM constraints will be compound of constraints obtained from the *Precondition Asserts*, *Postcondition Asserts* and *Source Code*. *Precondition Constraints* and *Postcondition Constraints* will directly be obtained from their formal specification. These constraints must necessarily be satisfied, because they express which are the initial and final conditions that a free of bugs source code must satisfy. In order to obtain the *Source Code Constraints*, we will divide the source code into basic blocks like : Sequential blocks (assignments and method calls), conditional blocks and loop blocks. Also, every block is a set of sentences that will be transformed into constraints.

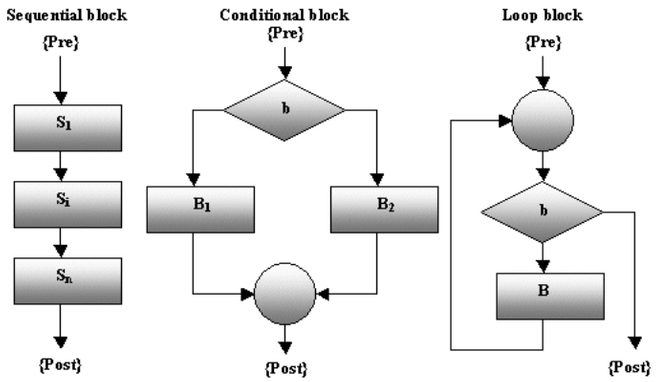

Fig. 2. Basic Blocks

- Sequential blocks: Starting from a sequential block as the one that appears in figure 2, we can deduce that the execution sequence will be: S_1...S_i...S_n. The

first step will be to *rename the variables*. We have to rewrite the sentences between the precondition and the postcondition in a way that will never allow two sentences to assign a value to the same variable. For example the code *x=a*c; ...x=x+3;...* {*Post:x =...* } would be transformed into *x1=a*c; ...x2=x1+3;...* {*Post:x2 =...* }.
Assignments: We will transform the source code assignments into equality constraints.
Method Calls: Our methodology only permits the use of methods calls that specify their precondition and postcondition. At present this specification is viable in object oriented languages as Java 1.4. For every method call, we will add the constraints defined in the precondition and the postcondition of this method to the PCM. When we find a recursive method call, this internal method call is supposed to be correct to avoid cycles in the diagnosis of recursive methods.
Our work is in progress in this point and there are still points that are being investigated. Due to it, we have to suppose that there are only functional methods (those that cannot modify the state of the object which contains the method declaration) and, also, these methods cannot return objects.

- Conditional blocks: We will often find a conditional block as it appears in the figure 2; we can deduce that the sequences will be :
  Sequence 1: {Pre }bB_1{Post} (condition b is true)
  Sequence 2: {Pre}¬bB_2{Post} (condition b is false)
  Depending on the test case, one of the two sequences will be executed. Therefore we will treat the conditional blocks as if they were two sequential blocks and we will choose one or the other depending on the test case. Then, we will transform it into constraints that will be part of the PCM. If we compare software diagnosis with the components diagnosis it would be as incorporating one or another component depending on the system evolution; this is something that has not been thoroughly treated in the components diagnosis theory. At this point we introduce improvements to our previous work [2], this methodology allows us to incorporate inequality constraints (in particular those which are part of the condition in the conditional sentences).
- Loop blocks: We will find loop blocks as it appears in figure 2. The sequences will be:
  Sequence 1: {Pre}{Post} (none loop is executed)
  Sequence 2: {Pre}bB_1{Post} (1 loop is executed)
  Sequence 3: {Pre}$b_1B_1b_2B_2...b_nB_n${Post} (2 or n loops are executed)
  Depending on the test case, one of the three sequences will be executed. To reduce the model to less than n iterations, and to obtain efficiency in the diagnosis process, we propose to add a sentence for each variable that would change value in the loop and would add the necessary quantity (positive or negative) to reach the value of the step *n-1*. The sequence 3 would be like:{*Pre*}$b_1\beta B_n${*Post*} where β will substitute $B_1 b_2 B_2 ... b_n$.
  For every variable X that changes its value in the loop, we will add the constraint X_{n-1}=$X_1+\beta_x$ what would allow us to maintain the value of X_n in the last step, and what would save us the *n-1* previous steps. The value of

β_x will be calculated debugging the source code. The constraints which add the β values cannot be a part of the diagnosis, because they are unaware of the original source code.

3.2 Obtaining the Minimal Diagnosis

Determining the Function to Maximize. The first step will be to define a set of variables R_i that allows us to perform a reified constraint model. A reified constraint will be like $C_i \Leftrightarrow R_i$. It consists of a constraint C_i together with an attached boolean variable R_i, where each variable R_i represents the truth value of constraint C_i (0 means false, 1 means true). The operational semantics are as follows: If C_i is entailed, then R_i=1 is inferred; if C_i is inconsistent, then R_i=0 is inferred; if R_i=1 is entailed, then C_i is imposed; if R_i=0 is entailed, then $\neg C_i$ is imposed.

Our objective is that most numbers of these auxiliary variables may take a true value. This objective will imply that we have to maximize the number of satisfied constraints. The solution search will be to maximize the sum of these variables, therefore the function to maximize will be: $\text{Max}(R_1+R_2+...+R_k)$.

Max-CSP resolution. Solving the Max-CSP we will obtain the set of sentences with a smaller cardinality, what caused the postcondition was not satisfied. To satisfy the postcondition we have to modify these sentences. To implement this search we used ILOG^{TM} Solver tools [6]. It would be interesting to keep in mind the works proposed in [8] and [4] to improve their efficiency in some problem cases.

4 Examples of Diagnosis

We have chosen five examples that show the grammar's categories to cover (a subset of the whole Java^{TM} language grammar). To prove the effectiveness of this methodology, we will introduce changes in the examples source code. With these changes the solution won't satisfy the postcondition. The diagnosis methodology should detect these changes, and it should deduce the set of sentences that cause the postcondition non satisfaction.

Example 1 : With this example we cover the grammar's part that includes the declarations and assignments. It will allow us to prove if the methodology is able to detect the dependencies among instructions. If we change the sentence S_5 by *g=y-z*, we will have a new program (named *example 1a*) that won't satisfy the postcondition. The assignments of the source code will be transformed into equality constraints. In example 1a the sentences S_1 to S_5 will be transformed into the result that appears in table 1. As we can observe, the methodology adds 5 equality constraints and the result is assigned in every case to a variable R_i which will be stored if the constraint is satisfied or not. These variables R_i will

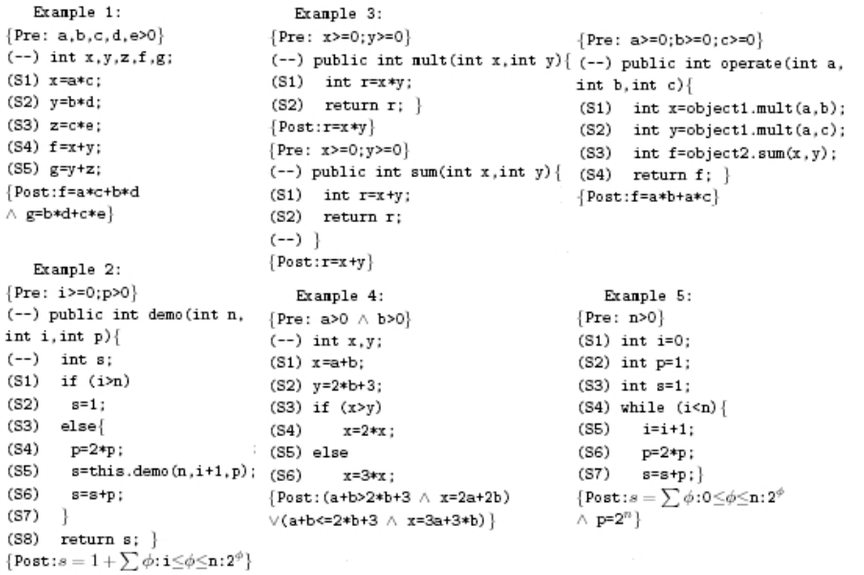

Fig. 3. Examples

be necessary to carry out the search Max-CSP to obtain the minimal diagnosis. These variables R_i will take the value 1 if the constraint is true or the value 0 if it is false.

Using a test case *TC*≡{*a=2,b=2,c=3,d=3,e=2,f=12,g=12*}, the obtained minimal diagnosis will include the sentence S_5 that is, in fact, the sentence that we had already changed; and also the sentence S_3. If we change S_3, it won't have any influence in S_4 but it will have influence in S_5, that is the sentence that we had changed. Therefore we will be able to return the correct result changing S_3, and without modifying S_5. It is necessary to emphasize that S_5 also depends on S_2, but a change in S_2 could imply a bug in S_4.

Example 2 : We will use example 2 to validate the diagnosis of recursive methods. We will change the sentence S_4 by *p=2*p+3*, with this change we will obtain the program *example 2a*. The PCM constraints of the examples 2a appear in table 1. The method calls are substituted by the constraints obtained of the postcondition of these method.

The variable R_3 (associated to the method call) should take the value 1 to avoid cycles in the recursive method diagnosis (as we explained in the previous section). In example 2a we will use the test case *TC*≡{*n=7,i=1,p=1*}, the diagnosis process reports us the sentences S_6 and S_4; the last one is, in fact, the sentence that we have changed. If we change S_6, we can modify the final result of *s* variable, and therefore, satisfy the postcondition with only one change.

Example 3 : This example will allow us to validate the diagnosis of non recursive methods. We will change the sentence S_2 by *y=object1.mult(b,c)* in *operate* method, obtaining the program that we will name *example 3a*. The PCM constraints of the example 3a appear in table 1. For example 3a, we show the constraints of *operate* method PCM. The method calls are substituted by the constraints obtained of the postcondition of those methods.

If we apply $TC \equiv \{a=2, b=7, c=3\}$ to the *operate* method (example 3a), we will obtain that the sentences S_1,S_2, and S_3 are the minimal diagnosis. The bug is exactly in S_2 because we called method with wrong parameters b and c instead of a and c. If we change the parameters that are used in the sentences S_1 and S_3, we can neutralize the bug in S_2. An interesting modification of example 3 would be to change the sentence S_3 by *f=sum(x,x)*. If we apply this change, the sentences S_1 and S_3 would constitute the diagnosis result. Now S_2 won't be part of the minimal diagnosis because sentence S_2 does not have any influence on the method result.

Example 4 : This example covers the conditional sentences. We have changed the sentence S_4 by *x=2*x+3*, and we will obtain the *example 4a*. If the inputs are a=6 and b=2, we can deduce that x>y; therefore S_4 will be executed. The result of the transformation of conditional sentence would be the constraint x>y and the transformation of sentence S_4 (in this occasion it is an assignment). We can see the result in table 1.

If we apply $TC \equiv \{a=7, b=2\}$ to example 4a we obtain the sentences S_1 and S_4 as a minimal diagnosis; this last one is in fact the sentence that we have changed. If we change S_1, we can modify the final result of x variable and, consequently, we will satisfy the postcondition. Therefore, it is another solution that would only imply one change in the source code.

Example 5 : We use this example to loop diagnosis. We will change the sentence S_7 by *s=2*s+p* and we will obtain the program *example 5a*. At this example the variables i, p and s change their values inside the loop. If i_0 is the value of i before the loop and i_{n-1} is the value of i in the step n-1, let's name β_i to the difference between i_{n-1} and i_0. Then the instruction $i_{n-1}=i_0+_{\beta i}$ (which will be before the loop) would allow us to conserve the dependence of the value i_n with previous values, and it would save us the *n-1* previous steps. The constraints which add the values β should not be part of the minimal diagnosis since they are unaware of the original source code. Therefore, the variables R_4,R_5 and R_6 must take the value 1 (as appears in table 1), this will avoid that they would be a part of the minimal diagnosis.

With $TC \equiv \{n=5, \beta_i=4, \beta_p=15, \beta_s=30\}$ we will obtain the sentence S_9 as minimal diagnosis. S_9 is exactly the sentence that we had already changed. The minimal diagnosis does not offer us S_{11} as minimal diagnosis because p takes a correct value (validated by the postcondition), although the value of s variable depends on the value of p variable. The problem is only in the s value, which does not satisfy the postcondition.

Table 1. PCM Examples

Example 1a

Precondition Constraints	Postcondition Constraints	Code Constraints	Observable Variables	Non observable Variables
a>0	f==a*b+b*d	R_1==(x==a*c)	a,b,c,d,e	x,y,z
b>0	g==b*d+c*e	R_2==(y==b*d)	f,g	
c>0		R_3==(z==c*e)		
d>0		R_4==(f==x+y)		
		R_5==(*g==y-z*)		

Example 2a

Precondition Constraints	Postcondition Constraints	Code Constraints	Observable Variables	Non observable Variables
i>=0	s1==1+	R_1==(i0<=n0)	n,i0,p0,	s1
p>0	$\sum \phi$:i0≤ϕ≤n:2^{ϕ}	R_2==(*p1==2*p0+3*)	p1,s0	
		R_3==(s0==1+		
		$\sum \phi$:(i0+1)≤ϕ≤n:2^{ϕ})		
		R_3==1		
		R_4==(s1==s0+p1)		

Example 3a

Precondition Constraints	Postcondition Constraints	Code Constraints	Observable Variables	Non observable Variables
a>0	f==a*b+a*c	R_1==(x==a*b)	a,b,c,	x,y
b>0		R_2==(*y==b*c*)	f	
c>0		R_3==(f==x+y)		

Example 4a

Precondition Constraints	Postcondition Constraints	Code Constraints	Observable Variables	Non observable Variables
a>0	(a+b>2*b+3 ∧	R_1==(x0==a+b)	a,b,x2	x0,x1,y0
b>0	x2=2a+2b) ∨	R_2==(y0==2*b+3)		
	(a+b<=2*b+3 ∧	R_3==(x0>y0)		
	x2=3a+3*b)	R_4==(*x2==2*x1+3*)		

Example 5a

Precondition Constraints	Postcondition Constraints	Code Constraints	Observable Variables	Non observable Variables
n>0	$s2 = \sum \phi$:0≤ϕ≤n:2^{ϕ}	R_1==(i0==0)	n,s2,p2,	s0,s1,p0,
	p2=2^n	R_2==(p0==1)	β_i,β_p,β_s,	p1,i0,i1,
		R_3==(s0==1)		i2
		R_4==(i0<n)		
		R_5==(i1==i0+ β_i)		
		R_6==(p1==p0+ β_p)		
		R_7==(s1==s0+ β_s)		
		R_5==R_6==R_7==1		
		R_8==(i2==i1+1)		
		R_9==(p2==2*p1)		
		R_{10}==(*s2==2*s1+p2*)		

5 Conclusions and Future Works

In this work we applied Max-CSP techniques to diagnose the software behavior. The explicit construction of the functional dependencies graph (proposed in other methodologies) has been avoided. We used only one TC to carry out the diagnosis, but we think that the use of a greater number of TC will improve our methodology to obtain software diagnosis. The investigation will continue in this line, looking for the way to include the result of several *TCs* to the diagnosis process of a same program. This will give us a more exact diagnosis. The final objective of our investigation is to extend the methodology to the complete grammar of an object-oriented language.

References

1. Robert V. Binder.: Testing Object-Oriented Systems: Models, Patterms, and Tools. Addison Wesley.
2. R. Ceballos, R. M. Gasca, Carmelo Del Valle y Miguel Toro: Diagnosis basada en modelos para la depuración de software mediante técnicas simbólicas. IV Jornadas de ARCA, Sistemas Cualitativos y Diagnosis, Vilanova i la Geltrú, Spain, June 2002.
3. Khalil, M.: Automated strategies for software diagnosis. The Ninth International Sympsosium on Software Reliability Engineering, Paderborn, Germany, Nov. 1998.
4. K. Kask.: New Search Heuristics for Max-CSP In Proceeding of CP'2000, pg. 262–277, 2000.
5. Khalil, M.: An Experimental Comparison of Software Diagnosis Methods. 25^{th} Euromicro Conference 1999.
6. ILOG: ILOG Solver 4.4 User's Manual. ILOG 1999.
7. J. Larrossa: Algorithms and Heuristics for Total and Partial Constraint Satisfaction. Ph.D dissertation, 1998.
8. J. Larrossa and P. Meseguer.: Partition-based lower bound for max-csp. Proceedings CP, pages 303–315, 1999.
9. Lyle J. R. and Weiser, M.: Automatic bug location by program slicing. Second International Conference on Computers and Applications, Beijing, China, pag. 877–883,June 1987.
10. Cristinel Mateis, Markus Stumptner, Dominik Wieland and Franz Wotawa.: Debugging of Java programs using a model-based approach. DX-99 Work-Shop,Loch Awe, Scotland (1999).
11. Cristinel Mateis, Markus Stumptner, Dominik Wieland and Franz Wotawa.: Extended Abstract – Model-Based Debugging of Java Programs. AADEBUG, August 2000, Munich.
12. Weiser, M.: Programmers Use Slices When Debugging. Communications of the ACM, Vol. 25, No. 7, pp.446-452,1982.
13. Weiser, M.: Program Slicing. IEEE Transactions on Software Engineering SE-10, 4, pp. 352–357, 1984