LOCALIZING BUGS IN PROGRAMS
*Or How to Use a Program's Constraint Representation for Software Debugging?**

Franz Wotawa
Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/2, 8010 Graz, Austria

Keywords: Fault localization, constraint-based reasoning.

Abstract: The use of a program's constraint representation for various purposes like testing and verification is not new. In this paper, we focus on the applicability of constraint representations to fault localization and discuss the underlying ideas. Given the source code of a program and a test case, which specifies the input parameters and the expected output, we are interested in localizing the root cause of the revealed misbehavior. We first show how programs can be compiled into their corresponding constraint representations. Based on the constraint representation we show how to compute root causes using constraint solver. Moreover, we discuss how the approach can be integrated with program assertions and unit tests.

1 INTRODUCTION

Localizing faults in software is generally considered a difficult and time consuming task. This holds especially in the case of software maintenance where the basic structure of the program and the underlying assumptions are not well understood or even not known. Although, there is a growing interest in fault localization within the research community, the overall problem is far away from being solved. One promising approach is Zeller's delta debugging (Zeller, 1999) applied to the isolation of cause-effect chains in programs (Zeller and Hildebrandt, 2002). But (Gupta et al., 2005) pointed out that from the cause-effect chain it is not always easy to really identify a fault and thus its applicability seems to be limited. Other approaches use slicing techniques like (DeMillo et al., 1996), (Kamkar, 1998), (Zhang et al., 2005) and others. Unfortunately, these approaches do no guarantee to remove all unnecessary parts of a program during a debugging session. For an overview on debugging we refer the reader to (Ducassé, 1993), (Shahmehri et al., 1995), and (Stumptner and Wotawa, 1998).

In this paper, we present an approach that is based on the syntax and semantics of a program. Thus the approach guarantees to focus on those parts of the program relevant in a certain debugging session. The approach requires that we have given a failure revealing test case, which specifies the input and the expected output of the program, and the source code of the program. For the sake of simplicity we assume that programs are written in a Java-like sequential programming language ignoring object-oriented features, multi-threading, and exceptions. The presented approach is based on model-based diagnosis (Reiter, 1987) and is most closely to (Ceballos et al., 2003; Ceballos et al., 2006).

The basic idea behind our approach is to compile the source code into a behavior equivalent constraint representation and to use this constraint representation directly to identify possible bug locations. Another underlying assumption is that the corrected program is a close variant of the given program. Hence, the approach is more suited for experienced programmers and less for programmers learning a programming language where a specialized tutoring system seems to be more suitable. Finally, we assume that the used test case is as small as possible and, therefore, does not require too many loop iterations or recursive procedure calls. This assumption is called the *small scope hypothesis*, which is used in other applications like verification (see (Jackson, 2006)).

So, why does a constraint representation of a program help to identify possible bug locations? Let us start with an analysis of the following statement *i*:

i. `x = a + 10;`

In *i* the value of variable `x` becomes the sum of the

*The work described in this paper has been supported by the FIT-IT research project *Self Properties in Autonomous Systems (SEPIAS)*, which is funded by the Austrian Federal Ministry of Transport, Innovation and Technology and the FFG.

Wotawa F. (2008).
LOCALIZING BUGS IN PROGRAMS - Or How to Use a Program's Constraint Representation for Software Debugging?.
In *Proceedings of the Third International Conference on Software and Data Technologies - SE/GSDCA/MUSE*, pages 88-95
DOI: 10.5220/0001890900880095

value of a and 10 during execution. The direction of computation is always from the inputs to the outputs. Hence, from a known value of x after the the execution of *i* we cannot derive a value for a using the programming language's semantics alone; although there is a relation between those variables. The constraint representation considers this relationship and straightforward line *i* is considered as relation between x and a. In our case this relation is a mathematical equation. Whenever x is known, a value for variable a can be derived and vice versa. But how does this help to focus on relevant parts of the program during debugging? To answer this question, we have a look at the following program:

```
1.  v = !in2;
2.  out1 = in1 && v;
3.  out2 = in3 || v;
```

Now consider the following test case: in1 = *true*, in2 = *false*, in3 = *false* for the inputs, and ou1 = *true*, out2 = *false* for the expected output. The program behaves incorrect with respect to the given test case. Instead of *false* the value *true* is computed for variable out2. When considering the data dependencies for variable out2 statements 1 and 3 might cause the misbehavior. Such a result would be returned when using slicing-based debugging approaches.

Consider now the relation-based representation of lines 1 to 3. We first, assume that Line 1 is incorrect. Therefore, we are not able to compute a value for target variable v. Because of Line 2 and the test case, we know that out1 has to be *true*. This can only be the case when both in1 and v are *true*. Hence, we are able to determine the value of v. From this value we obtain out2 to be *true*, which contradicts the test case. The same happens when assuming Line 2 to be faulty. The only remaining candidate is Line 3, which improves the result. One reason for the improvement is that relations allow for reasoning in all possible directions and not only from the inputs to the outputs.

In the rest of the paper we introduce the compilation of programs into their constraint representation. For this purpose we first convert programs into programs where all loops and recursive procedure or method calls are unrolled. From this representation we extract its static single assignment (SSA) form. The SSA form can be easily mapped to a set of constraints. We further present an algorithm, which allows for computing all statements that might cause the misbehavior, and show how the approach can be easily combined with program assertions and testing.

2 CONVERSION – PART 1

Within the paper, we assume that programs are written in a Java like programming language ignoring object-oriented features, multi-threading, and exceptions. Moreover, we ignore all type specific information and assume that the use of functions, variables, and other language constructs, which require type conformity, is done in a type-safe fashion. The first part of the conversion comprises two steps. In the first step, we convert all not necessarily recursive procedures and while statements of the program using unrolling of sub-blocks and procedure bodies. In the second step, we convert the resulting program into its SSA form. In the SSA form every variable is only used once as a target. Finally, the second part of conversion uses the SSA form to obtain a constraint representation.

We do not formally specify the conversion process but explain the necessary steps and discuss important issues. The overall idea of using constraints in software engineering is not new. However, most of the research activities focus on verification except (Ceballos et al., 2003; Ceballos et al., 2006), where the constraint representation is used for fault localization. (Gotlieb et al., 1998) used constraints for test case generation. More recently, (Collavizza and Rueher, 2006) introduced the conversion of programs into constraints and used them for verification purposes.

2.1 Recursion and Iteration

Under some restrictions every program comprising recursion and/or loop statement can be converted into a loop free but behavior equivalent form. The restriction applies to the program's input requiring that the input is chosen in a way where the maximum number of recursive calls or iterations is known in advance. This is of course not possible in the general case and seems to be somehow awkward or very restrictive. However, when considering a program that runs in its environment similar restrictions apply. For example, because of memory constraints the number of recursions is limited. Moreover, like in (Jackson, 2006) we argue that the test cases used to validate a program are usually small and require not too many loop iterations or recursive functions. Since, unrolling loops and recursive calls increases the overall program size of the resulting program, a restriction on the maximum number of iterations or recursions is necessary. The unrolling step is necessary for debugging in order to make all iterations explicit, which allows for a direct integration of loop and recursion invariants. The invariants only need to be copied in every unrolled

block. Moreover, in principle the number of iterations and recursive calls can be determined from the failure revealing test case directly.

We use the following rules in order to compute the recursion free and loop free variants of a given program. We handle recursive procedure as well as loops by unrolling the involved statements. This unrolling of statements is done up to a pre-specified bound. To allow for detecting that a test case reaches this boundary, we use a new variable `fail`. `fail` is initialized with the value *false*. If the number of required iterations exceed the given parameter, the variable `fail` is set to *true*.

Recursion. Given a procedure call `y = p(a1,...,an)` at line *i* of the program, and the declaration of the procedure, which comprises the formal parameters $x_1, \ldots, x_n$ and the set of statements *S*. For simplicity, we assume that the body of `p` comprises only one return statement at the end of the program. We construct a new body S', which is equivalent to *S* but where the return statement of the form `return e;` is replaced with `return_p = e;`. The conversion is done by replacing the procedure call with the following statements in the case where the maximum number of considered recursive calls is not reached.

x_1 `= a1; ...`x_n `= an;` S' `y = return_p;`

Note that if there is no return value, the last statement can be ignored and S' is equivalent to *S*. If the pre-defined maximum number of recursive call is reached, the call is replaced by a signal assignment `fail = true;` where the variable `fail` is used only for stating that something unexpected happened, and shall not be used in the original program. Note that the maximum number of recursive calls limits the recursion depth. For example, if there are two calls of the same recursive procedure in a block, then the maximum number of recursions for both is the same.

While/Loops. A while statement of the form `while` *C* { *S* } can be easily converted into a nested if-structure. Every time the condition *C* evaluates to *true* the statements in *S* are executed and testing *C* is done again, until *C* evaluates to *false*. Hence, in general we can replace a while statement by the following infinite structure of nested if-statements.

`if` *C* { *S* `if` *C* { *S* `...`} } }

In practice we have to set the nesting depth similar to the maximum number of recursion when converting recursive procedures. If the maximum nesting depth is reached, we add the statement `if` *C* { `fail = true;` }.

With these two conversions rules we replace all recursive and non-recursive functions and while statements with their equivalent structures. The obtained program comprises more statements than the original program but does not contain loops anymore.

The unrolling of recursive functions and loops is different from the original conversion of programs into their equivalent SSA form, which does not require an unrolling. However, in our case the unrolling allows for explicitly considering every single iteration step during debugging. The SSA form, which we use for debugging, is a SSA form of the converted program without loops and recursions and not a SSA form of the original program. Since, the unrolling does not change the semantics of the program if the variable `fail` is not set to `true` during program exectution using the failure revealing test case, the SSA form of the converted program is also semantically equivalent to the original program under the same conditions. This ensures the correct computation of diagnoses.

2.2 Static Single Assignment Form

The SSA form of a program (Cytron et al., 1991) is a representation with the property that no two left-side variables share the same name. Hence, every variable that is defined in a statement has a unique name. The SSA form of a program is of importance in our case because it can be directly mapped to constraints. We discuss this issue later and briefly introduce the conversion into an SSA form. For more information regarding the SSA form and its computation we refer to (Cytron et al., 1991; Aycock and Horspool, 2000; Mayer, 2003) where also the conversion of arrays and other data structures is explained.

The conversion of programs into their SSA form can be done by adding an index to every variable. A variable that is used obtains the index from the last definition of the same variable. Every time a variable is defined a new index is generated. If a program comprises only assignment statement, the conversion is straightforward. In case of conditional statements or loops the conversion becomes more complicated. In our case, we only have to consider conditional statements because the loop statements and the recursive procedure calls are eliminated in the previous conversion step.

The idea behind the conversion of conditional statement is as follows: The value of the condition is stored in a new unique variable. The if- and the else-branches are converted separately. In both cases the conversion starts using the indices of the variables already computed. Both conversions deliver back new indices of variables. In order to get a value for a variable we have to select the last definition of a variable

```
1.  if (state == 1) {
2.    if (on) {
3.      state = 2;
4.    }
5.  else if (state == 2) {
6.    if (off) {
7.      state = 1;
8.    }
9.  }
10. if (state == 1) {
11.   v = !in2;
12.   out1 = in1 && v;
13.   out2 = in3 || v;
14. }
15. if (state == 2) {
16.   out1 = false;
17.   out2 = false;
18. }
```

Figure 1: A small example program `fsm`.

from the if- and else-branch depending on how the if condition evaluates. This selection is done using a function Φ, which is defined as follows:

$$\Phi(x,y,C) = \begin{cases} x & \textbf{if } C = true \\ y & \textbf{otherwise} \end{cases}$$

Hence, for every variable which is defined in the if- or the else-branch we have to introduce a selecting assignment statement, which calls the Φ function.

Let `if` C `{.. x = .. } else { .. x = .. }` be a conditional statement at line n of the program. The SSA form is given as follows:

```
var_n = C;
...  x_i = ...
...  x_j = ...
x_k = Φ(x_i,x_j,var_n);
```

The indices i, j, and k of `x` are assumed to be the indices assigned to `x` in order to meet the properties of the SSA form.

We illustrate the conversion using the program `fsm`, that implements a small finite state machine. Such programs often occur in the embedded systems domain, which is one of the target domains of our approach. The program comprises 1 state variable `state`, 5 input variables `on, off, in1, in2, in3`, and 2 output variables `out1, out2`. Lines 1-9 implement the state transitions as a function of `on, off`, and lines 10-18 the output function, which specifies values for `out1, out2` as a function of `in1, in2, in3` and the internal state `state`.

The SSA form of `fsm` is depicted in Figure 2. It comprises only assignment statements. All conditional statements are replaced by assignment statements where the right-hand side calls the Φ function

```
1.  var_1 = (state_0 == 1);
2.  var_2 = (on_0);
3.  state_1 = 2;
4.  state_2 = Φ(state_1,state_0,var_2);
5.  var_5 = (state_0 == 2);
6.  var_6 = (off_0);
7.  state_3 = 1;
8.  state_4 = Φ(state_3,state_0,var_6);
9.  state_5 = Φ(state_4,state_0,var_5);
10. state_6 = Φ(state_2,state_5,var_1);
11. var_11 = (state_6 == 1);
12. v_1 = !in2_0;
13. out1_1 = in1_0 && v_1;
14. out2_1 = in3_0 || v_1;
15. out1_2 = Φ(out1_1,out1_0,var_11);
16. out2_2 = Φ(out2_1,out2_0,var_11);
17. var_15 = (state_6 == 2);
18. out1_3 = false;
19. out2_3 = false;
20. out1_4 = Φ(out1_3,out1_2,var_15);
21. out2_4 = Φ(out2_3,out2_2,var_15);
```

Figure 2: The SSA form of program `fsm`.

for each variable used as target in either the then-branch or else-branch.

Before discussing the conversion of programs in SSA form to constraints, we introduce the basic concepts of constraint systems including constraint solving.

3 CONSTRAINTS

In order to be self contained, we briefly discuss the basic definitions of constraint systems including the computation of solutions. For a more in-depth presentation of constraint systems and their algorithms we refer to (Dechter, 1992), (Mackworth, 1987), and (Dechter, 2003). A constraint system *CS* is characterized by a set of variables $V = \{V_1, \ldots, V_n\}$, each of them associated with a (not necessarily finite) domain D_i, $1 \leq i \leq n$, and a set of constraints $C = \{C_1, \ldots, C_k\}$. Each of the constraints C_j has a corresponding pair (X_j, R_j), where $X_j \subseteq V$ is a set of variables, and R_j is a relation over X_j. X_j is called the scope of constraint C_j. For convenience we assume a function $dom : V \mapsto DOM$ that maps a variable $V = i$ to its domain D_i, a function $scope : C \mapsto 2^V$ that maps a constraint to its corresponding scope, and a function $rel : C \mapsto RELATIONS$ that maps constraints to their relations.

An assignment of values to all variables is called an instantiation. An instantiation is said to be ful-

filled, if it does not contradict any constraint. A constraint is said to be in contradiction with an instantiation iff the variable values are not represented in the constraint relations. Usually someone is interested in finding non-contradictory, i.e., fulfilling, instantiation, which we also call a solution. An effective way in practice for computing solutions is to use backtrack search. For backtracking we assume an ordered variable collection VO. We start with the first variable and assign a provisional value. We further assign provisional values to the successive variables as long as the constraints are fulfilled. For this purpose we only have to consider constraints where all variables have an assigned value. If one constraint is violated we backtrack to the variable that has been assigned a value in the last step and choose another value. If there is no value, we have again to track back to the previous variable and so on. If there is no further value to assign for the first variable, there is no solution. Otherwise, the procedure stops when all variables have been assigned values that fulfill all constraints.

The following algorithms implements backtracking and has to be accessed via the call **findSolution(CS,VO,∅)**. It returns a solution if one exists. Otherwise, the algorithm returns the empty set. The algorithm only guarantees to terminate on constraint systems where all domains are finite. This is the case in debugging given a failure revealing test case where the test case can be used to restrict the the domains.

findSolution(CS,VO,*I*)

1. If VO is empty, then return the current variable assignment I as result.
2. Otherwise, let v be the first element of VO.
3. For all values $x \in dom(v)$ of the currently selected variable v do:
   (a) Add the assignment $v = x$ to the set of current assignments I.
   (b) Check all constraints where variables have a value assignment in I. If at least one constraint is violated, remove $v = x$ from I. Otherwise, do the following:
      i. Call **findSolution(CS,VO**$\setminus\{x\}$**,*I*)** recursively and store the result in r.
      ii. If $r = \emptyset$, then remove $v = x$ from I. Otherwise, return r.
4. Return $\emptyset$.

Beside optimizations regarding the used data structures, there are three ways for improving the running time of the backtracking algorithm, i.e., variable ordering (see (Freuder, 1982; Dechter and Pearl, 1989)), restricting domains, and detecting dead ends during search as fast as possible (see (Dechter and Pearl, 1988)).

4 CONVERSION – PART 2

Programs in SSA form have a simple structure, comprise only assignment statements, and every variable is defined only once. Hence, the conversion is easy and requires only a conversion of each statement separately. All variables are mapped to corresponding variables of the constraint systems. The data types of the variables are mapped to the domains of the constraint variables. Each statement is mapped to a constraint where the corresponding constraint variables of variables used in the statement form the scope of the constraint. The constraint's relation is given by the statement itself.

For example, Line 14 of the SSA form of program `fsm` (Figure 2) `out2_1 = in3_0 || v_1;` is converted in a constraint C_{14} with $scope(C_{14} = \{out2\_1, in3\_0, v\_1\}$ and relation $rel(C_{14}) = \{out2\_1 = in3\_0 \vee v\_1\}$. The relation of C_{14} has to be interpreted as a logical rule where '=' is the bi-implication and '$\vee$' a logical or. For finite domains the relation can also be represented in a tabular form.

$out2\_1$	$in3\_0$	v_1
$false$	$false$	$false$
$true$	$true$	$false$
$true$	$false$	$true$
$true$	$true$	$true$

Accordingly to our definition of consistency of constraints, a given instantiation, i.e., an assignment of values to constraint variables is consistent with a constraint, if the corresponding tuple is element of the relation. Otherwise, the instantiation contradicts/does not fulfill the constraint. For C_{14} $out2\_1 = false$, $in3\_0 = true$, $v\_1 = false$ is an inconsistent instantiation. If changing the value of $in3\_0$ to $false$, we obtain a consistent instantiation.

The conversion of statements to the corresponding relation is rather straightforward and has to be defined for the functions and predicates of the data types used in the program. Because of space limitations we will not consider all of these conversions. Instead we discuss the conversion of assignments comprising the Φ function. Assume a statement $x\_i = \Phi(x\_j, x\_k, c)$ in line n of the program. We map this statement to a corresponding constraint C_n with $scope(C_n) = \{x\_i, x\_j, x\_k, c\}$. The relation of C_n is specified using

3 rules: $c = true \rightarrow x\_i = x\_j$, $c = false \rightarrow x\_i = x\_k$, $x\_j = x\_k \rightarrow x\_i = x\_j$. The latter rule states that if the value of the variable x is the same for both branches of a conditional statement, then it can be used after the execution of the branches.

The second part of the compilation process, i.e., the conversion of the SSA form into a set of constrains, does obviously not change the behavior. Every result of a program run of the SSA form with respect to the given input values is a (partial) solution of the corresponding constraint system using the same input values and vice versa. Note that in cases where the number of chosen iterations is not enough, the used `fail` variables are set to *false*. In this case, the SSA form and its corresponding constraint representation do not reflect the behavior of the original program anymore. A solution is to perform the compilation process again with an increased number of allowed iterations.

Until now, we are not able to use the constraint representation together with the backtracking algorithm to compute possible fault candidates, i.e., statements that cause misbehavior. The reason is that we are not able to distinguish the correct behavior of a statement from its incorrect one. A solution is to explicitly state correct and incorrect behavior. To do so, we extend the constraints by introducing a new status variable S_n for each constraint C_n. The domain of S_n is $\{ok, ab\}$ where *ok* represents the correct and *ab* the incorrect, i.e., abnormal, behavior. The relations obtained from the statements are mapped to the correct behavior. This can either been done via replacing a rule r with $S_n = ok \rightarrow (r)$ in the constraint's relation or by adding a new column to the tabular form and setting the value of S_n to *ok* for each row.

In addition, we have to specify the faulty behavior. Since, the real faulty behavior is not known, we assume the following fault model: *A faulty assignment statement does not allow to determine a value for the target/defined variable. All other variables are not influenced.* In order to implement this fault model, we introduce a new value ?, which represents 'don't know'. The ? is assumed to be element of all domains, which belong to the domains of corresponding program variables. For the rule representation of a constraint C_n with target variable x, we add the rule $S_n = ab \rightarrow x =$?. In tabular form we add a new column where ? is assigned to all variables except S_n, which has to be equivalent to *ab*. Moreover, a further improvement would be to add new rules even for the correct behavior. For example, a logical or would be *true* if only one arguments is *true*.

For constraint C_{14} we state the extended behavior in tabular form as follows:

$out2\_1$	$in3\_0$	v_1	S_{14}
false	*false*	*false*	*ok*
true	?	*true*	*ok*
true	*true*	?	*ok*
?	?	?	*ab*

In addition to these changes, we slightly adapt the term consistency of constraints with respect to variable instantiations. We say that a constraint is consistent with a given instantiation, iff there exists a tuple in the constraint relation that can be mapped to the instantiation. A tuple can be mapped to an instantiation iff there exists a replacement of of each ? value for a variable with an element of the domain that makes the tuple equivalent to the instantiation. Note that not all ? have to be replaced with the same value. Every variable instantiation can be mapped to the faulty behavior. This ensures that at least one explanation can always be found. From here on, we always assume that a program to be debugged is compiled into a constraint representation that allows for debugging.

5 FAULT LOCALIZATION

Given the constraint representation $CS_\Pi = (V, D, C)$ of a program Π and a test case T_Π revealing a faulty behavior, the backtracking algorithm **findSolution** can be used in order to determine the cause of the misbehavior. The following steps are necessary for this purpose:

1. Let VO be the variable ordering where the first elements are the status variables $S_1, \ldots, S_{|C|}$ of the constraints $C_i \in C$, followed by the variables, which are specified in the test case T_Π, and the remaining variables from V.
2. For each input of the form $x = v$ in T_Π add a new constraint with scope $\{x\_0\}$ and relation $x\_0 = v$ to C.
3. For each expected output in T_Π of the form $y = v$ add a new constraint with scope $\{y\_i\}$ and relation $y\_i = v$ to C where i is the greatest index of variable y.
4. Call **findSolution(**(V, D, C)**,** VO**,** $\emptyset$**)** and return the result.

Note that this algorithm returns only one solution. It can be adapted in order to find all or a pre-defined number of solutions by adapting **findSolution** accordingly. In a practical setting the algorithm has to be adapted in order to first search for single fault candidates and afterwards for multiple fault candidates. But it is important to consider that the approach is not limited to single faults.

When applying the approach to the small example program given in the introduction of this paper, we receive as solution that Statement 3 is faulty. The corresponding instantiation is $in1\_0 = true, in2\_0 = false, in3\_0 = false, v\_1 = true, out1\_1 = true, out\_2 = true, S_1 = ok, S_2 = ok, S_3 = ab$.

Besides handling multiple faults the proposed approach allows for an easy integration of assertions and unit tests. Assertions are basically nothing else than conditions that are evaluated during runtime. If the condition evaluates to false the assertion is said to fail. Otherwise, the assertion is said to be fulfilled. For example, an assertion for the `fsm` program in Figure 1 would specify the state transitions:

```
    ⋮
5.  else if (state == 2) {
6.    if (off) {
7.      state = 1;
8.    }
9.  }
    @ASSERT[on = true ⇒ state = 2]
    @ASSERT[off = true ⇒ state = 1]
10. if (state == 1) {
11.   v = !in2;
    ⋮
```

The first assertions specifies that whenever `on` is set to true, the `state` variable should be 2. The second assertions specifies that `off` changes `state` to 1. A implicit assumption behind this assertions is that either `on` or `off` are true. The integration of the assertions is easy. They only have to be converted into constraints using the variable indices at the given location. Note that assertions are not allowed to change variables. Hence, we do not need to take care of new variable indices. For our example program, we would add the following constraints to the `fsm`'s constraint representation:

on_0	$state\_6$
$true$	2

off_0	$state\_6$
$true$	1

For both constraints no status variables are added, because we assume that assertions cannot fail.

The integration of unit tests can be done in a similar way. Unit test usually have the following structure. At the beginning the variables are set to their initial values. Then the program is called. Finally, assertions are used to specify the expected values. In case one assertion is contradicted, an exception is raised and the unit test is assumed to fail. We illustrate the integration of unit tests using the `fsm` program again.

```
    @INIT[state = 2 ∧ on = false ∧ off = true]
    @INIT[in1 = true ∧ in2 = false ∧ in3 = false]
1.  if (state == 1) {
2.    if (on) {
    ⋮
17.   out2 = false;
18. }
    @ASSERT[out1 = true ∧ out2 = false]
```

For integration purposes we again convert the initialization and the assertion to constraints, which are assumed to be correct.

$state\_0$	on_0	off_0
2	$false$	$true$

$in1\_0$	$in2\_0$	$in3\_0$
$true$	$false$	$false$

$out1\_4$	$out2\_4$
$true$	$false$

The integration of assertions and unit tests as shown is smooth when using a constraint representation. Moreover, the approach puts the computation of fault candidates down to the computation of solutions for the corresponding constraint representation. For the latter there are efficient algorithms available. Hence, for smaller programs up to 500 statements locating bugs should be possible. We implemented the compilation of programs into constraints and tested the approach on very small programs using a self-implemented constraint solver. The results in terms of the number of diagnosis candidates are promising but require an improvement on side of the constraint solver as well as some optimizations during the conversion. In particular, time for computing fault candidates has to be improved substantially in order to be of use in practical applications. At the moment diagnosis time is between a fraction of a second and about 1 minute for small programs comprising 10 to 20 lines of code.

6 CONCLUSIONS

There are many different approaches for fault localization but most of them are based on data flow and control flow. In this paper, we presented an approach that is based on the constraint representation of a program and a failure revealing test case for computing fault candidates. The advantage of constraints is the availability of constraint solver, which can be directly used. The research described in the paper is most closely to the application of model-based diagnosis (see (Reiter, 1987)) to software debugging as described in (Köb and Wotawa, 2006) and (Ceballos et al., 2003; Ceballos et al., 2006).

In contrast to previous research the presented approach offers: (1) An almost standardized way of representing programs as constraints; (2) Debugging is put down to constraint solving where a lot of research is devoted to constraint solving algorithms; (3) The integration of assertions and unit tests can be easily done in our case. There is no need for a special treatment. Assertions can be added to the compiled program on the fly during a debugging session; And (4) The debugging results depends on the syntax and the semantics of a programming language.

Of course the complexity of debugging is still high and improvements of both the solving algorithms and the conversion process are necessary. The handling of object-oriented features, multi-threaded programs, and exception are still open issues. However, in specialized areas like the embedded systems domain, the application of the presented approach is in reach.

REFERENCES

Aycock, J. and Horspool, N. (2000). Simple generation of static-single assignment form. In *Proceedings of the 9th International Conference on Compiler Construction (CC)*, pages 110–124.

Ceballos, R., Casca, R. M., Valle, C. D., and Borrego, D. (2006). Diagnosing errors in dbc programs using constraint programming. In *Selected Papers from the 11th Conference of the Spanish Association for Artificial Intelligence (CAEPIA 2005)*, volume 4177 of *Lecture Notes in Computer Science*.

Ceballos, R., Gasca, R., Valle, C. D., and Rosa, F. D. L. (2003). A constraint programming approach for software diagnosis. In Ronsse, M. and Bosschere, K. D., editors, *Proceedings of the Fifth International Workshop on Automated Debugging*, Ghent, Belgium.

Collavizza, H. and Rueher, M. (2006). Exploration of the capabilities of constraint programming for software verification. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 182–196. Springer, Vienna, Austria.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490.

Dechter, R. (1992). Constraint networks. In *Encyclopedia of Artificial Intelligence*, pages 276–285. Wiley and Sons.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Dechter, R. and Pearl, J. (1988). Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38.

Dechter, R. and Pearl, J. (1989). Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366.

DeMillo, R. A., Pan, H., and Spafford, E. H. (1996). Critical slicing for software fault localization. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 121–134.

Ducassé, M. (1993). A pragmatic survey of automatic debugging. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, AADEBUG '93*, Springer LNCS 749, pages 1–15.

Freuder, E. C. (1982). A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32.

Gotlieb, A., Botella, B., and Rueher, M. (1998). Automatic test data generation using constraint solving techniques. In *Proc. ACM ISSTA*, pages 53–62.

Gupta, N., He, H., Zhang, X., and Gupta, R. (2005). Locating faulty code using failure-inducing chops. In *Automated Software Engineering (ASE)*, pages 263–272.

Jackson, D. (2006). *Software abstractions: logic, language, and analysis*. MIT Press.

Kamkar, M. (1998). Application of program slicing in algorithmic debugging. *Information and Software Technology*, 40:637–645.

Köb, D. and Wotawa, F. (2006). Fundamentals of debugging using a resolution calculus. In Baresi, L. and Heckel, R., editors, *Fundamental Approaches to Software Engineering (FASE'06)*, volume 3922 of *Lecture Notes in Computer Science*, pages 278–292, Vienna, Austria. Springer.

Mackworth, A. (1987). Constraint satisfaction. In Shapiro, S. C., editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. John Wiley & Sons.

Mayer, S. (2003). Static single-assignment form and two algorithms for its generation. Seminar Work, Winter Term 2002/03, University of Konstanz, http://www.inf.uni-konstanz.de/dbis/teaching/ws0203/pathfinder/download/ mayers-ausarbeitung.pdf.

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

Shahmehri, N., Kamkar, M., and Fritzson, P. (1995). Usability criteria for automated debugging systems. *J. Systems Software*, 31:55–70.

Stumptner, M. and Wotawa, F. (1998). A Survey of Intelligent Debugging. *AI Communications*, 11(1).

Zeller, A. (1999). Yesterday, my program worked. today, it doesn't. why? In *Proceedings of the Seventh European Software Engineering Conference/Seventh ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 253–267.

Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2).

Zhang, X., He, H., Gupta, N., and Gupta, R. (2005). Experimental evaluation of using dynamic slices for fault localization. In *Sixth International Symposium on Automated & Analysis-Driven Debugging (AADEBUG)*, pages 33–42.