ECAI 2008
M. Ghallab et al. (Eds.)
IOS Press, 2008

doi:10.3233/978-1-58603-891-5-797

From constraint representations of sequential code and program annotations to their use in debugging[1]

Mihai Nica and **Franz Wotawa**[2]

1 Introduction

Debugging, i.e., the detection, localization, and correction of bugs, has been considered an important task in software engineering. A lot of research has been devoted to debugging but mainly to fault detection. In this paper we focus on fault localization, which is based on the constraint representation of programs. For this purpose, programs are converted into their equivalent constraint satisfaction problem (CSP). A solution of the corresponding CSP is a diagnosis candidate. Besides the source code, a failure revealing test case has to be given. For more information regarding CSP we refer to [2].

The work described in this paper is most closely to the work of Ceballos et al. authors [9], where constraint programming is used for fault localization. There approach requires that the programmer provides *contracts*, i.e., pre- and post-conditions, for every function. However, the authors do not investigate the complexity of solving the resulting problem and the scalability to larger programs. In particular, they do not consider structural decomposition or other methods for improving constraint solving, which would make the approach feasible.

In order to complement previous research, we investigate the complexity of solving the CSP corresponding to a debugging problem that comprises the source code and the test case. In the past, in order to find problem classes which are tractable, much work has been done on the structural decomposition of CSPs. Gottlob et al. proposed the *hypertree decomposition*, and they showed that this decomposition method generalizes other important methods [3, 4]. The *hypertree width*, a characteristic of the structure of the constraint system, is a measure for the complexity of solving a CSP and, therefore, a measure for the complexity of the debugging problem. In other words, by performing a hypertree decomposition we can obtain a metric for the complexity of debugging.

2 Example

In this section we use a small example program to motivate fault localization using constraint-based reasoning with integrated annotations. For the program in Figure 1 assume that Line 3 is changed to '`while (i <= x) {`' which leads to an obviously wrong implementation. If we are only interested in finding single faults at the statement level, we use the following process. Statement by statement we go through the program and assume the current statement to be faulty. All other statements are considered to work as expected.

```
    { x ≥ 0 ∧ y ≥ 0 } // PRE-CONDITION
1.  i == 0;
2.  r == 0;
3.  while (i < x) {
    { r == i · y } // INVARIANT
4.    r = r + y;
5.    i = i + 1;
    }
    { r == x · y } // POST-CONDITION
```

Figure 1. A program for computing the product of two natural numbers

When assuming a statement to be faulty, we cannot derive a value for those variables defined in the statement. A variable is said to be defined within a statement if a value is assigned to the variable. Such a semantics for faulty statements is implemented in the previous model-based diagnosis approaches of debugging, e.g., in [5].

We now assume that Line 1 of the multiplication program behaves faulty. In this case the variable `i` in Line 1 is assigned the undefined value `?`, that means: 1.`i` =?;. Because of this change, we are not able to decide whether the condition in Line 3 evaluates to true or false. Hence, no values for `r` or `i` can be determined, and finally, we cannot contradict the expected value. As a consequence Line 1 is a valid diagnosis accordingly to model-based diagnosis [6]. The same happens when assuming Line 2 to be faulty. In this case `r` has no value assigned. From the other information we know that the sub-block of the while is executed once. Hence, we receive the following equation, where the available information is given in parentheses:

4. {r =? ∧ y=2} `r = r + y;` {r=0}

This equation can be solved by setting the value of `r` (before executing the statement) to `-2` which does not contradict the value `?`. A similar situation occurs for the other statement, and hence, there is no way of excluding even a single statement from the list of possible bug candidates. This problem of not being able to exclude statements is due to the fact of missing information. In order to overcome this problem we have to combine verification information and debugging. For this purpose we consider program annotations which can also be used for verification based on Hoare's calculus like the one given in Figure 1. When now using the same procedure for finding single faults, only Lines 1 and 3 remain as diagnosis results. We now prove that Line 2 can be excluded and it is easy to see that the same argument applies to Line 4 and 5 as well. If assuming Line 2 to be faulty, we receive the following equation:

```
{ r == i · y ∧ i==1 ∧ x==0 ∧ y==2}
```

[1] This research has been funded in part by the Austrian Science Fund (FWF) under grant P20199-N15 and by the FIT-IT research project *Self Properties in Autonomous Systems(SEPIAS)* which is funded by BMVIT and the FFG
[2] Technische Universität Graz, Institute for Software Technology, 8010 Graz, Inffeldgasse 16b/2, Austria, {mihai.nica,wotawa}@ist.tugraz.at. Authors are listed in alphabetical order.

```
4.   r = r + y;
     {r==0}
```

From `i==1` and `i==0` we derive `r==0` before executing the statement. Hence, we obtain `r` to be 2 after the execution which contradicts the expected value of `r`. Statement 2 is no single fault diagnosis anymore.

This simple example shows that the integration of verification information that is based on program annotations really improves debugging. Hence,a representation of programs and there annotations as constraints together with a constraint solver can be used to check correctness assumptions of program statements.

3 Debugging process

The whole conversion algorithm of programs into their equivalent CSP representation and its use in debugging is described in [10]. We only briefly discuss the overall diagnosis process that comprises the following steps:

1. *Remove loops*: The first step is to remove all while statements and recursive function calls by 'unrolling'. For this purpose a while statement is converted into a nested if-statement. A similar procedure is done for recursive functions. Since, the maximum number of iterations is known for a given test case, the resulting loop-free program behaves in the same way like the original program.
2. *SSA Conversion*: In the second step, the loop-free program is converted into its static single assignment (SSA) form. In the SSA form every variable is defined once. For more information regarding the SSA form we refer to [1]. In this step the assertions are also converted.
3. *The CSP's hyper-tree*: From the SSA form we build the constraints system and its corresponding hyper-tree. This is done by mapping every program variable to its corresponding constraint variable. Every assignment is mapped directly to a constraint. The behavior of the constraints is given by the semantics of the corresponding statements.
4. *Diagnosis*: In the diagnosis step, we use the resulting CSP and the given test case directly for solving the obtained debugging problem. For this purpose we use the TREE* algorithm [7]. The algorithm requires an acyclic CSP, which can be obtained by applying for example hyper-graph decomposition [3, 4] or other decomposition methods. The combination of TREE* and composition method is described in [8].

When using the debugging process, the complexity of debugging is equivalent to the complexity of solving a CSP. [4] states that the complexity of solving a CSP is related to the hyper-tree width of the CSP as follows: The time need it to find a solution for a CSP with n variables as input and a corresponding hyper-tree width of k is in the worse case $O(n^k \log n)$. Hence, knowing the hyper-tree width of CSPs of programs is important in practice. In Table 2 we report first results regarding the hyper-tree width of some small programs comprising while- and if-statements. The table comprises the lines of code (LOC), the lines of code of the corresponding SSA form (LOC2), the number of while-statements (#W), the number of If-statements (#I), the number of considered iterations (#IS), and the hyper-tree width (HW) for each program. The hyper-tree width of the programs varies from 3 to more than 30, which indicates that computing diagnosis candidates is a complex task when relying on CSP representation of programs. Another important issue is that the hyper-tree width increases when the number of considered iterations (during the unrolling step) increases. Whether there is an upper-bound or not is still an open issue.

Name	LOC	LOC2	#W	#I	#IS	HW
BinSearch	27	40	1	3	1	3
BinSearch	27	112	1	3	4	8
Binomial	76	82	5	1	1	3
Binomial	76	1155	5	1	30	≥ 30
Hamming	27	62	5	1	1	2
Hamming	27	989	5	1	10	≥ 14
Huffman	64	78	4	1	1	2
Huffman	64	342	4	1	20	≥ 12
whileTest	60	88	4	0	1	2
whileTest	60	376	4	0	9	8
Permutation	24	41	3	1	1	3
Permutation	24	119	3	1	7	6
Permutation	24	1231	3	1	100	6
Adder	63	70	0	5	-	3
SumPowers	21	33	2	1	1	2
SumPowers	21	173	2	1	15	10
SumPowers	21	1376	2	1	100	10
IscasC432	162	162	0	0	-	9
ComplexHypertree	12	30	1	0	1	3
ComplexHypertree	12	370	1	0	30	17
ComplexHypertree	12	1076	1	0	100	17

Figure 2. The hyper-tree width for different sequential programs

4 Conclusions

In this paper we discussed the compilation of programs into their equivalent CSP representation and its use for fault localization. Assertions like pre- and post-conditions or invariants can be easily integrated. Moreover, CSP solvers can be directly used for debugging.

Solving CSPs depends also on their structural properties. The structural properties of the CSP corresponding to a given problem, represent an indicator for the complexity of program debugging.

In this paper, we give first results of the debugging complexity using hyper-tree width. The results show that debugging requires a lot of computational resources.

REFERENCES

[1] Marc M. Brandis and H. Mössenböck. Single-pass generation of static assignment form for structured languages. *ACM TOPLAS*, 16(6):1684–1698, 1994.
[2] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
[3] Georg Gottlob, Nicola Leone, and Francesco Scarcello. On Tractable Queries and Constraints. In *Proc. DEXA 2001*, Florence, Italy, 1999.
[4] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *AI*, 124(2):243–282, 2000.
[5] Wolfgang Mayer, Markus Stumptner, Dominik Wieland, and Franz Wotawa. Can ai help to improve debugging substantially? debugging experiences with value-based models. In *ECAI*, pages 417–421, Lyon, France, 2002.
[6] Raymond Reiter. A theory of diagnosis from first principles. *AI*, 32(1):57–95, 1987.
[7] Markus Stumptner and Franz Wotawa. Diagnosing tree-structured systems. *AI*, 127(1):1–29, 2001.
[8] M. Stumptner and F. Wotawa. Coupling CSP decomposition methods and diagnosis algorithms for tree-structured systems. In *Proc. 18th IJCAI*, pages 388–393, Acapulco, Mexico, 2003.
[9] R. Ceballos and R. M. Gasca and C. Del Valle and D. Borrego Diagnosing Errors in DbC Programs Using Constraint Programming *Lecture Notes in Computer Science*, Vol. 4177, Pages 200-210, 2006.
[10] Paper waiting to be reviewed by Informatica. *http://www.informatica.si/*