On the Complexity of Program Debugging Using Constraints for Modeling the Program's Syntax and Semantics

Franz Wotawa[1], Jörg Weber[1], Mihai Nica[1], and Rafael Ceballos[2,⋆]

[1] Institute for Software Technology
Graz University of Technology
[2] Computer Languages and Systems Department
University of Seville

Abstract. The use of model-based diagnosis for automated program debugging has been reported in several publications. The quality of the obtained results in terms of debugging accuracy is good. Unfortunately, most of the proposed models and techniques have very high computational needs. In this paper we focus on giving an explanation for the high computational needs of debugging. In particular, we propose a constraint representation of programs whose behavior is equivalent to the original programs. We further analyze the constraint representation to obtain its hypertree width, which is an indicator for the complexity of the corresponding constraint satisfaction problem. As constraint-based debugging is equivalent to constraint solving, the hypertree width is also an indicator for the debugging complexity. We further show that it is possible to construct arbitrarily complex programs such that their hypertree width is not bounded as indicated in previous literature.

1 Introduction

During programming and, even worse, during maintenance activities, locating and fixing faults in the source code is an arduous and difficult task, which is hardly automated in today's integrated development environments. One reason is the limited applicability of currently published approaches due to the underlying assumptions. E.g., approaches using frequency measures are only appropriate if a large test suite is available and if a tight integration of test case execution and debugging is given. Approaches that rely on the program's semantics provide good results even in cases where only one test case is available but they suffer from high computational requirements, which prevent them of being used for larger programs. Other approaches that are based on the program's structure can be used for larger programs but usually exhibit bad discrimination among possible fault candidates.

⋆ This research has been funded in part by the Austrian Science Fund (FWF) under grant P20199-N15 and is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT). Authors are listed in reverse alphabetical order.

P. Meseguer, L. Mandow, and R.M. Gasca (Eds.): CAEPIA 2009, LNAI 5988, pp. 22–31, 2010.

When considering the current state of research in debugging, should we conclude that current approaches to automated debugging have been not successful? The answer to this question is definitely no! There are many methods for automated debugging which prove that automation is possible and effectively helps focusing on those parts of a program which are more likely to be faulty. E.g., Peischl and Wotawa [12] provided empirical evidence that model-based diagnosis [14] can be effectively used to reduce the number of statements that someone has to consider during debugging. The published results indicated a median reduction of 97 percent for debugging sequential hardware designs. Mayer and colleagues [5] presented results on the use of model-based diagnosis for debugging of sequential programs. The work has been extended to handle object-oriented languages and the use of program abstraction; see Mayer's PhD thesis [9].

In this paper we tackle the problem of giving reasons for the high computational demands of automated debugging, in particular when using a model-based diagnosis approach which utilizes the complete semantics of the program in order to obtain precise diagnosis.

Obviously, in general model-based diagnosis itself requires a lot of computational resources, but a reduction is possible when focusing on single faults only. However, even in this case the debugging of larger programs using the complete semantics requires a lot of computing time. In this paper we explain why this is the case. By mapping programs to their equivalent constraint representation we reduce the debugging problem to a constraint satisfaction problem. Constraint representations of programs have been described in literature for different purposes like verification [13] and also for debugging [11]. In contrast to verification which deals with fault detection, debugging deals with fault localization and correction.

The tree width of the constraint satisfaction problem, which can be obtained from the hypertree representation, is an indicator of the complexity for computing solutions. The accuracy of this metric is especially high when the constraints are extensionally modeled. We show that it is possible to construct arbitrarily complex programs such that their hypertree width is not bounded. Moreover, we present experimental results that indicate a high tree width for debugging problems.

2 Representing Programs as Constraints

In this section we describe the conversion of programs into their constraint representation. We also show the equivalence of both with respect to a given test suite. In particular, we show that if a program computes an output O for a certain input I, then its constraint representation will also compute the same values for the corresponding output variables.

Our work addresses sequential programs with a syntax and semantics similar to well-known languages like Java, but without object-oriented constructs. For simplicity we do not consider procedure calls in this paper, but it should be noted that procedures can be straightforwardly integrated as shown in [10]. Our approach supports assignments statements, conditional statements, and loops.

Throughout the rest of this paper we use the program given in Fig. 1 as running example.

```
1.  i = 0;
2.  r = 0;
3.  while (i < x) {
4.    r = r + y;
5.    i = i + 1;
    }
```

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

```
1.  i = 0;
2.  r = 0;
3.  if (i < x) {
4.    r = r + y;
5.    i = i + 1;
    }
```

Fig. 2. Loop unrolling of the program from Fig. 1 for 1 iteration

The first step for obtaining a constraint representation of an arbitrary program is the conversion into a loop-free variant. The second step is to obtain a static single assignment form. The third step takes the static single assignment form and maps it to a constraint system. In order to prove that the conversion step does not change the program's behavior we have to define what we mean with program equivalence. First of all, converted program should compute the same outputs for the given inputs. Moreover, the names of the variables in the converted program may differ from the original program, but there is a correspondence between the variables, and so the corresponding variables must be compared. Since we want to use the representation for debugging, we restrict the program equivalence to the given test cases:

Definition 1. *Given a program Π and a program Π', an input environment $\omega_I \in ENV$, a set of relevant output variables $OUTPUT \subseteq VARS$, and a function $\sigma : VARS \mapsto VARS$ mapping variables from Π to the corresponding variables in Π'. We say that the programs Π and Π' behave equivalent if and only if for all relevant output variables $x \in OUTPUT$ the same values are computed, i.e., $\omega_O(x) = \omega'_O(\sigma(x))$, where $\omega_O = I(\Pi, \omega_I)$ and $\omega'_O = I(\Pi', \omega'_I)$ with $\omega'_I(\sigma(v)) = \omega_I(v)$ for all variables $v \in VARS$. We write $\Pi =_{\omega_I} \Pi'$.*

Note that all variables that are not specified in $OUTPUT$ might have different values after applying the interpretation function. We extend Def. 1 to a set of input environments of the available test suite:

Definition 2. *Two programs Π and Π' are equivalent if they are equivalent for all input environments ω_I, i.e., $\forall\, \omega_I : \Pi =_{\omega_I} \Pi'$. In this case we write $\Pi = \Pi'$.*

2.1 Loop-Free Programs

Since loops cannot be directly represented as a constraint system, we transform the original program to a loop-free variant and prove its equivalence.

If the body of a $while$-loop is executed at most once, then the behavior corresponds to the single execution of a conditional statement. In general, if a $while$-condition is fulfilled, the statements in the block are executed and afterwards the condition is evaluated again. Hence, programs can be compiled into their loop-free equivalent if the number of steps is known in advance. As we debug a program using a given test case, we can simply execute the program for this test case in order to determine the maximum number of iterations.

Example: The loop-free variant of the program from Fig. 1 is depicted in Fig. 2 for $n = 1$ iteration. It can be used for all cases where $x \in \{0, 1\}$ without leading to a different behavior compared to the original program.

Because of the loop unrolling the size (i.e., the number of statements) of the resulting loop-free program increases. The amount of increase depends on the time complexity of the original program. This results from the fact that the runtime of a program directly corresponds to the number of statement executions. The more iterations a program has, the more statements are executed. We summarize this finding in the following theorem:

Theorem 1. *Given a program Π with time complexity $O(f(\Sigma))$ where Σ denotes the size of the input. The corresponding loop-free program Π_{LF} has to have a size of $O(f(\Sigma_M))$ where Σ_M is the maximum input size to be considered.*

As a consequence, using this model directly for debugging is only feasible for programs which have a low time complexity or when using inputs of smaller size. Although the focus of this paper is on the complexity of debugging and not on the practicability of the approach, we briefly discuss how to tackle the challenge of feasibility. One way would be to compute the loop-free program in such a way that the number of nested if-statements does not exceed the number of iterations in the given test cases. Then it is guaranteed that the model captures the whole test suite. An improvement would be to consider only failing (negative) test cases and to further reduce the test suite to small test cases.

2.2 Static Single Assignment Form

The constraint representation requires that all left-side variables in the program have unique names. Hence, we use an intermediate representation of the program, the Static Single Assignment (SSA) form, which has the property that no two left-side variables share the same name [1]. Since all variables are defined only once, the SSA form allows for a clear representation of the dependencies that are established between different variables inside the corresponding program.

Unique variable names can be easily obtained by adding an index at the end of the name. However, although converting programs comprising only assignment statements is straightforward, it is more difficult to convert programs with loops or conditional statements. As we transform loops to nested if-statements, we only need to consider conditional statements.

The idea behind the conversion of conditional statements 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 from the if- and $else$-branch depending on how the if condition evaluates. This selection is done using a function `phi`. Hence, for every variable which is defined in the if- or the $else$-branch we have to introduce a selecting assignment statement which calls `phi`.

For example, the corresponding SSA form of the program fragment `if` e `{.. x = ..} else { .. x = ..}`
is given as follows:

```
var_e = e; ...; x_i = ...; x_j = ...;
x_k = phi(x_i,x_j,var_e);
```

where the function phi is assumed to be part of the used data type and is defined as follows:

$$\texttt{phi}(x, y, b) = \begin{cases} x \text{ if } b = TRUE \\ y \text{ otherwise} \end{cases}$$

The SSA representation for the program from Fig. 1 is depicted in Fig. 3. For brevity, only one iteration is considered.

```
1.  i_0 = 0;
2.  r_0 = 0;
3.  var_e_4 = (i_0 < x_0);
4.  r_1 = r_0 + y_0;
5.  i_1 = i_0 + 1;
6.  r_2 = phi(r_1,r_0,var_e_4);
7.  i_2 = phi(i_1,i_0,var_e_4);
```

Fig. 3. The SSA form of the loop-free variant of the program from Fig. 1 (for one iteration)

Obviously the transformation of loop-free programs into their SSA form does not have an influence on the actual behavior (apart from the variable renaming), i.e., $\Pi_{LF} = \Pi_{SSA}$ without any restrictions.

Lemma 2. *Given a program Π, a number $n \in \mathbb{N}$, a set of relevant output variables $OUTPUT \subseteq VARS$, and a function σ, which is defined as follows: For all variables $x \in VARS \setminus OUTPUT$: $\sigma(x) = x\_0$. For all output variables $y \in OUTPUT$: $\sigma(y) = y\_M(y)$, where $M(y)$ is the largest index assigned to a variable y. The loop-free variant $\Pi_{LF} = \Gamma(\Pi, n)$ of Π behaves equivalent to its SSA form Π_{SSA}. I.e., for all input environments $\omega_I \in ENV : \Pi_{LF} =_{\omega_I} \Pi_{SSA}$.*

Note that in this lemma we implicitly assume, for simplicity, that output variables cannot be used as inputs for a program. This assumption does not contradict generality.

2.3 Constraint Representation

The final step of the conversion is the compilation to a constraint satisfaction problem (CSP). A CSP (V, D, CO) is characterized by a set of variables V, each variable having a set of domains D, and a set of constraints CO, where each constraint defines a relation R between variables. The variables occurring in a relation $R \in CO$ are called the scope S_R of the relation. A solution of a CSP is an assignment of values to all variables which does not contradict any given constraint. For more information regarding CSPs we refer to Dechter [2].

The *extensional modeling of a constraint* is the explicit representation of all allowed combinations of values for the variables of a constraint, within a certain domain. For example, if we have the boolean equation $z = y \wedge x$, $D_x = D_y = D_z = \{0, 1\}$, then the extensional representation of this constraint is given by the following tuples: $< 0, 0, 0 >, < 0, 0, 1 >, < 0, 1, 0 >, < 1, 1, 1 >$.

An extensionally modeled CSP has the advantage that it can be tractable based on its structure(i.e., hypergraph) rather than on the relations between its constraints. Based on the structural representation of the CSP, we can determine the complexity of finding its solution. Our framework implies such a modeling of the constraints.

The constraint representation of a program is extracted from its SSA representation.Let Π_{SSA} be a program in SSA form. Then the corresponding CSP $CSP(\Pi_{SSA})$ is constructed as follows:

- All variables in Π_{SSA} are variables of the CSP.
- The domain of the variables in the CSP is equivalent to the datatype.
- Every statement $x = e$ can be converted to a relation R where the scope $\{x_1, \ldots, x_n\}$ is equivalent to the set of variables used in expression e. The relation $R(x, x_1, \ldots, x_n)$ is defined as follows: For all $\omega \in ENV$ with $\omega(x_i) = v_i$: if $I(x = e, \omega) = \omega'$, then $R(\omega'(x), v_1, \ldots, v_n)$ is true, otherwise false.

Lemma 3. *Given a program Π_{SSA} in SSA form and its corresponding CSP representation $CSP(\Pi_{SSA})$. For all $\omega \in ENV$: $I(\Pi_{SSA}, \omega) = \omega'$ iff $\omega \cup \omega'$ is a solution of $CSP(\Pi_{SSA})$.*

Using this lemma we can finally conclude that the whole conversion process is correct:

Theorem 4. *Given a program Π and $n \in \mathbb{N}$, the loop-free representation for n iterations, the SSA form and the CSP representation of Π are equivalent under the given assumptions, i.e., $\Pi = \Pi_{LF} = \Pi_{SSA} = CSP(\Pi)$.*

This theorem follows directly from Lemma 1 to 3.

Example: From the SSA form which is depicted in Fig. 3 we extract the following CSP representation:

Variables: $V = V_N \cup V_B$ with
$V_N = \{x\_0, y\_0, i\_0, r\_0, r\_1, i\_1, r\_2, i\_2\}$ and
$V_B = \{var\_e\_4\}$

Domains: $D = \{D(x) = \mathbb{N} \mid x \in V_N\} \cup \{D(x) = \{TRUE, FALSE\} | x \in V_B\}$

Constraints:
$$CO = \left\{ \begin{array}{l} i\_0 = 0, r\_0 = 0, var\_e\_4 = (i\_0 < x\_0), r\_1 = r\_0 + y\_0, i\_1 = i\_0 + 1, \\ r\_2 = phi(r\_1, r\_0, var\_e\_4), i\_2 = phi(i\_1, i\_0, var\_e\_4) \end{array} \right\}$$

3 Constraint-Based Debugging

In the previous section we introduced a model that captures the syntax and semantics of an arbitrary program. The model itself is a constraint system which can be used for debugging by applying model-based diagnosis [14] techniques. For this purpose we introduce the predicate $AB(s_i)$, which means that statement s_i is abnormal. The constraints are changed to the form $\neg AB(s_i) \rightarrow C(s_i)$, where $C(s_i)$ is the original constraint corresponding to statement s_i. E.g., the constraint $(i\_0 = 0)$ is changed to $\neg AB(s_1) \rightarrow (i_0 = 0)$, where s_1 represents statement 1 in the SSA-form in Fig. 3.

Based on the debugging model we are able to compute diagnosis. Intuitively, a diagnosis is a subset of statements s.t. the assumption that those statements are abnormal and that all other statements are not abnormal does not contradict a given test case. Note that test cases are also represented as constraints (without the AB predicate).

Hence, debugging is reduced to CSP solving and so we can use techniques from CSP solving to classify the debugging complexity. It is well known that CSPs whose corresponding hypergraph is acyclic can be solved in polynomial time [2]. A hypergraph is a graph where edges connect an arbitrary number of vertices. The hypergraph for a CSP represents variables as vertices and the constraint scopes as edges. Moreover, a cyclic hypergraph can be decomposed into an acyclic one using available decomposition methods [3]. The resulting acyclic hypergraph is a tree structure where each vertex represents several constraints which have to be joined in order to solve the CSP. The maximum number of constraints to be joined is the tree width. When using the hypertree decomposition, which is the most general decomposition method [3], the width is called hypertree width. The hypertree width is a very good indicator for the complexity of solving a extensionally modeled CSPs: as stated in [3], any given CSP can be solved in $O(|\Sigma|^k * log|\Sigma|)$, where k is the hypertree width and Σ the input size of the CSP.

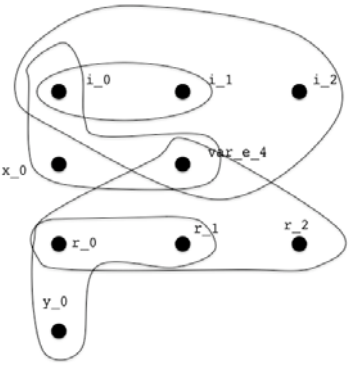

Fig. 4. The hypergraph of the debugging model for the program in Fig. 1

The hypergraph of the debugging model of the program in Fig. 1 is depicted in Fig. 4. In the following we discuss the consequences of the debugging model in terms of complexity. In particular we are interested whether the hypertree width is bounded for such a model or not. In [7] the author proved that structured programs have a hypertree width of 6 in the worst case. Unfortunately, the result is based on considering the control flow graph only but not the data flow, which is sufficient for some tasks to be solved in compiler construction. The following theorem shows that the result of [7] cannot be applied in the context of debugging where the control and data flow is of importance.

Theorem 5. *There is no constant upper-bound for the hypertree width of arbitrary programs.*

Proof: (Sketch) We prove this theorem indirectly. We assume that there is a constant value which serves as upper-bound for all programs and show that there is a class of

programs where this assumption does not hold. Consider the class of programs that is obtained running the following algorithm for $n > 0$:

1. Let Π_n be a program comprising an empty block statement.
2. For $i = 1$ to n do:
   (a) For $j = i + 1$ to n do:
      i. Add the statement
      $$x_{j,i} = x_{i,i-1} + x_{j,i-1}$$
      at the end of the block statement of program Π_n.
3. Return Π_n.

In this class programs have n inputs and 1 output. Every variable depends on any other directly or indirectly via another variable. Hence, the hypertree width depends on the number of statements and there is no constant value, which contradicts our initial assumption. □

Note that there is always an upper-bound of the hypertree width of a particular program, which is given by the number of statements. However, the above theorem states that there is no constant which serves as upper-bound for all programs. What is missing is a clarification whether the number of nested if-statements after loop-unrolling has an influence on the hypertree width. If the hypertree width of a program comprising a *while*-statement depended on the number of considered iterations, then the complexity of debugging would heavily depend on the used test cases. Fortunately this is not the case as stated in the following theorem.

Theorem 6. *Given an arbitrary program Π comprising at least one* while *statement. There always exists an upper bound on the hypertree width when the number of iterations increases.*

Proof: (Sketch) If we have a *while* statement, we obtain a sequence of block statements each block representing an iteration of the execution. These blocks are connected via variables defined in the block statement of the while and used in it. Hence, the sequence can be easily mapped to a tree where each node comprises all statements of the while's block statement. Hence, the hypertree width of this tree is bounded by the number of statements in the block statement. Variables that are defined outside the *while* statement and used by a statement inside do not increase the hypertree width because they are part of every node. Only the variables that are defined within the while's block statement have to be looked at more closely. These variables are summarized to a single variable using the `phi` functions. What we have to do is to add the corresponding constraint to each node. Because this has to be done only once per node the overall hypertree width of the resulting structure is bounded. Hence, the overall tree representing the *while* statement is bounded by the number of statements used in its block statement. □

4 Experimental Results

As mentioned, the hypertree width is an important metric for the complexity of debugging based on a constraint representation. Complex debugging problems have a large hypertree width. In general, problems with a hypertree width of more than 5 are hard problems.

For computing the hypertree and the hypertree width we relied on an implementation provided by [6] which employs the Bucket Elimination algorithm [8]. Note that this algorithm is an approximation algorithm, i.e., it does not always generate the optimal hypertree decomposition with a minimal width. However, as reported in [8], the algorithm which performs the optimal decomposition is very time and space demanding and is therefore not suitable for practical use, and the Bucket Elimination algorithm in most cases provides better approximations than other known approximation algorithms.

The obtained results are summarized in Fig. 5. For each program the table comprises the lines of code (LOC) of the original program (column *P*) and of the corresponding SSA form (*S*), the number of $while$-statements (*#W*), the number of if-statements (*#I*), the max. number of considered iterations (*#IS*) and the hypertree width (*HW*) with its minimum (*min*) and maximum (*max*) value. In *min* we depict the hypertree width for one iteration and in *max* the hypertree width for the max. number of iterations as stated in column *#IS*.

	#LOC					HW	
Name	P	S	#W	#I	#IS	min	max
BinSearch	27	112	1	3	4	3	8
Binomial	76	1155	5	1	30	3	30
Hamming	27	989	5	1	10	2	14
ArithmeticOp	12	12	0	0	-	1	1
Huffman	64	342	4	1	20	2	12
Multiplication	10	350	1	0	50	2	5
whileTest	60	376	4	0	9	2	8
Permutation	24	119	3	1	7	3	6
ComplexHT	12	370	1	0	30	3	17
SumPowers	21	173	2	1	15	2	10
ATestCase	43	682	4	4	10	5	7
BM4bitPAV2_F1	21	21	0	0	-	2	2
BM4bitPAV2_F2	21	21	0	0	-	2	2
IscasC17_F3	6	6	0	0	-	2	2
IscasC17_F1	6	6	0	0	-	2	2
BM4bitAdder_F1	26	26	0	0	-	4	4
IscasC432_F1	160	160	0	0	-	9	9

Fig. 5. The hypertree width for different sequential programs

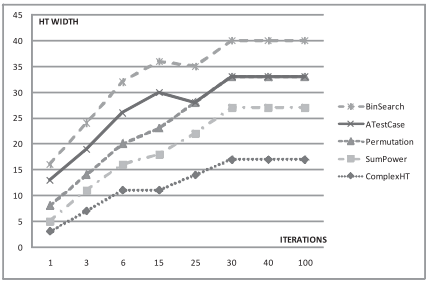

Fig. 6. The hypertree width as a function of the number of iterations when unrolling $while$-statements. For the sake of clarity we depict only 5 programs from the set of tested programs.

It can be seen that the hypertree width varies from 1 to more than 30. Although the obtained results are only for small programs they allow us to conclude that debugging programs is a very hard problem from the computational point of view.

The hypertree width obviously depends on the number of unrollings. We performed several experiments with a larger number of iterations. Fig. 6 depicts the hypertree evolutions of five different programs. It can be seen that in all of these cases the hypertree width reaches an upper bound, as indicated in Theorem 6.

5 Conclusion

Debugging is considered a computationally very hard problem. This is not surprising, given the fact that model-based diagnosis is NP-complete. However, more surprising is that debugging remains a hard problem even when considering single-faults only, at least when using models which utilize the entire semantics of the program in order to obtain precise results. In this paper we identified the hypertree width as an explanation. We have shown that it is possible to generate arbitrarily complex programs. So there is no constant upper bound for the hypertree width of programs. For programs with loops, the number of iterations to be considered for debugging has an impact on the hypertree width. This impact is limited by our finding that, for a given program, there is an upper bound on the hypertree width which is independent from the number of iterations.

We provided empirical results using small well-known programs like binary search. The computed values for the hypertree width varied from 1-30. Even the trivial Multiplication program, which has only 10 lines of code, has a hypertree width of 5 when considering 50 iterations. These values are very high, which explains why debugging is such a hard problem. A practical contribution of our work is the reduction of debugging to a constraint satisfaction problem, which allows for the use of standard CSP solvers for debugging.

References

1. Brandis, M.M., Mössenböck, H.: Single-pass generation of static assignment form for structured languages. ACM TOPLAS 16(6), 1684–1698 (1994)
2. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)
3. Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural CSP decomposition methods. Artificial Intelligence 124(2), 243–282 (2000)
4. DeMillo, R.A., Pan, H., Spafford, E.H.: Critical slicing for software fault localization. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 121–134 (1996)
5. Mayer, W., Stumptner, M., Wieland, D., Wotawa, F.: Can ai help to improve debugging substantially? debugging experiences with value-based models. In: Proceedings of the European Conference on Artificial Intelligence, Lyon, France, pp. 417–421 (2002)
6. `http://www.dbai.tuwien.ac.at/proj/hypertree/index.html`
7. Thorup, M.: All Structured Programs have Small tree width and Good Register Allocation. Information and Computation Journal
8. Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B., Musliu, N., Samer, M.: Heuristic Methods for hypertree Decompositions, DBAI-TR-2005-53, Technische Universität Wien (2005)
9. Mayer, W.: Static and Hybrid Analysis in Model-based Debugging. PhD Thesis, School of Computer and Information Science, University of South Australia, Adelaide, Australia (2007)
10. Wotawa, F., Nica, M.: On the Compilation of Programs into their equivalent Constraint Representation. Informatica 32(4), 359–371 (2008)
11. Ceballos, R., Gasca, R.M., Del Valle, C., Borrego, D.: Diagnosing Errors in DbC Programs Using Constraint Programming. In: Marín, R., Onaindía, E., Bugarín, A., Santos, J. (eds.) CAEPIA 2005. LNCS (LNAI), vol. 4177, pp. 200–210. Springer, Heidelberg (2006)
12. Wotawa, F., Peischl, B.: Automated source level error localization in hardware designs. IEEE Design and Test of Computers 23(1), 8–19 (2006)
13. Collavizza, H., Rueher, M.: Exploring Different Constraint-Based Modelings for Program Verification. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 49–63. Springer, Heidelberg (2007)
14. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32(1), 57–95 (1987)