Using Model-Based Reasoning for Locating Faults in VHDL Designs

Markus Stumptner and Franz Wotawa

Model-based reasoning has seen successful application in hardware oriented and physical domains and many models of physical devices are currently available. In the software domain is still ongoing work. In this paper we describe two kinds of models for programs which are used for locating faults in VHDL designs, suitable for embedding into the VHDL development process and with the actual system description automatically derivable from the code of the program under scrutiny.

Introduction

Today's hardware designers commonly use dedicated hardware description languages such as VHDL for describing the behavior of their designs. Since VLSI designs are becoming larger and larger, there is an increasing need for tools that support detecting misbehaviors and locating the responsible parts of the program. Currently, misbehaviors are detected by comparing the output of a design in form of signal value changes over time, i.e., a waveform trace (WFT), with a specification trace (see Figure 1) or by comparing the whole design to a formal specification. The first type of detection is called validation, while the latter is known as verification. Once a misbehavior has been detected, the designer will be interested in those parts of the program which caused the program to fail. This could be, e.g., individual statements, processes, or modules.

In the past, different approaches have been proposed for the basic task of helping a programmer find and fix bugs. Some of them use dependencies between variables and signals [10][2][1], typically working directly on the basis of the program code but requiring more user interaction to find the bug. Other approaches use the program semantics (e.g.,[5][8]). All those listed have in common that they do not need an additional specification or user interaction to built a tool specific data structure from the program code. However, some of the approaches use special-purpose algorithms or rely on a specific programming language. In this paper we describe the application of model-based reasoning to software debugging to overcome this limitation. We discuss two different models, one based exclusively on dependency information, and the other using the full semantics of VHDL programs. However, both models can be applied using the same diagnostic algorithm.

Both models, described in the next section, can be automatically derived from the source code of a given VHDL program, eliminating the need for writing a separate formal specification. Nonetheless we need some kind of knowledge about the expected behavior of the program to perform any kind of diagnosis. This behavior is given in terms of expected values for signals occurring in the program. The values are specified by the user in terms of value sequences over time of a set of input-output vectors. These sequences may stem from different sources: a verbal specification that the hardware designer translates into an actual test set, output data obtained from an earlier high-level version of the design that is assumed to be correct (the most common situation), or output of a verification tool which returned a counter-example to show that some desired property does not hold.

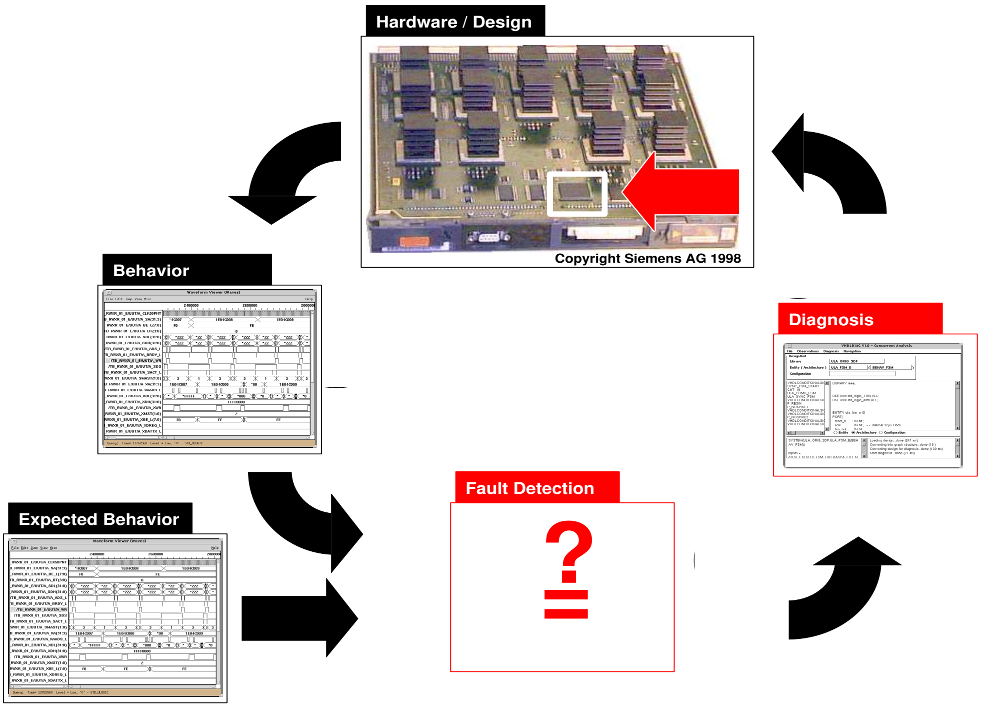


Figure 1 Software debugging cycle

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity COUNTER is
  port(
        E1,E2,CLK: in STD_LOGIC;
        A1,A2,A3,A4: out STD_LOGIC);
end COUNTER;

architecture BEHAV of COUNTER is
  signal D1,D2,Q1,Q2,NQ1,NQ2: STD_LOGIC := '0';
begin

  -- Input combinatorial block
  comb_in: process (Q1,Q2,E1,E2)
    variable I1,I2: STD_LOGIC;
  begin
    I1 := not((Q1 and Q2) or (not(Q1) and not(Q2)));
    I2 := (I1 and E1) or (not(I1) and not(E1));
    D1 <= (E1 and E2) or (E2 nor I2);
    D2 <= (E1 and E2) or (E2 nor Q2);
  end process comb_in;

  -- Output combinatorial block
  comb_out: process(Q1,Q2,NQ1,NQ2)
  begin
    A1 <= NQ2 and NQ1;
    A2 <= Q2 and NQ1;
    A3 <= NQ2 and Q1;
    A4 <= Q1 and Q2;
  end process comb_out;

  NQ1 <= not(Q1);

  NQ2 <= not(Q2);

  -- Memory block
  dff: process(CLK)
  begin
    if ((CLK = '1') and CLK'EVENT then
      Q1 <= D1;
      Q2 <= D2;
    end if;
  end process dff; [0.1em]

end BEHAV;
```

Figure 2 A VHDL counter implementation

Although we believe that model-based diagnosis is general enough to handle almost all fault situations occurring for software, we restrict our model to deal with functional bugs, e.g., misused variables or functions. Syntax errors, errors leading to runtime exceptions, and bugs causing the program to loop infinitely often are out of the scope of the models and this paper. In addition, we assume that the correct program is only a small variant of the program to be debugged. This distinguishes our work from work in the field of tutoring systems [3].

Modeling for VHDL debugging

The crucial decision when building a diagnosis model from program code involves the choice of those parts of the program which are supposed to be represented as components in the model. This decision will naturally directly influence the granularity of the diagnosis results. If statements are mapped to components, bugs are localized on the statement level, and faults inside a statement cannot be diagnosed without fault modes, i.e., descriptions of how a particular type of fault changes the system behavior. (In the software domain, fault modes can be expressed as repair actions on a program, see [6] for a more detailed discussion.) On the other hand, using expressions as diagnosis components allows finer granularity, but at the cost of significantly increasing the number of components and the diagnosis time.

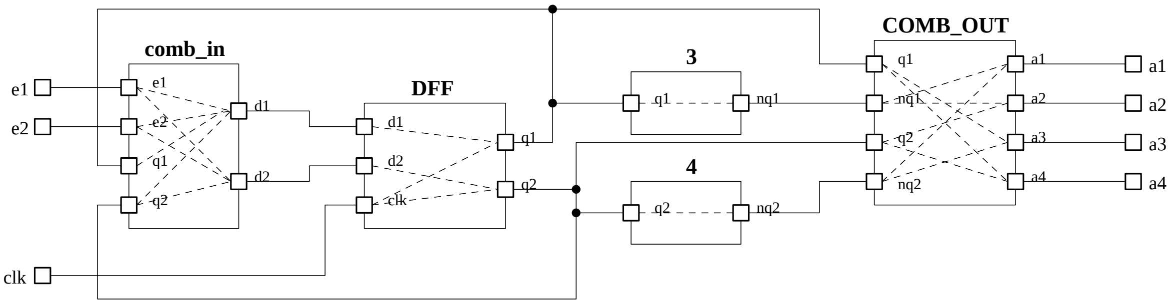
Apart from the granularity of diagnostic results, one also has to decide on how to represent the domain of variables in the program. For the purpose of this paper we restrict the discussion to scalar values. We can either use the full domain given by the type of a variable or signal occurring in a program, or use a more abstract representation.

We use the VHDL implementation of a simple counter (see Figure 2) to illustrate how models can be derived from programs. The behavior of the counter is specified as follows. If both inputs are set to 0, the counter increments its value. If E1 is set to 1 and E2 to 0, the value is decremented. If E1 is set to 0 and E2 to 1 the counter is set to 0, and in the last case where both inputs are set to 1, the counter is set to 1. The output is coded using a 1-from-4 decoder. A counter value 0 is represented by the output vector A1=1, A2=0, A3=0, A4=0, and 3 is represented by A1=0, A2=0, A3=0, A4=1.

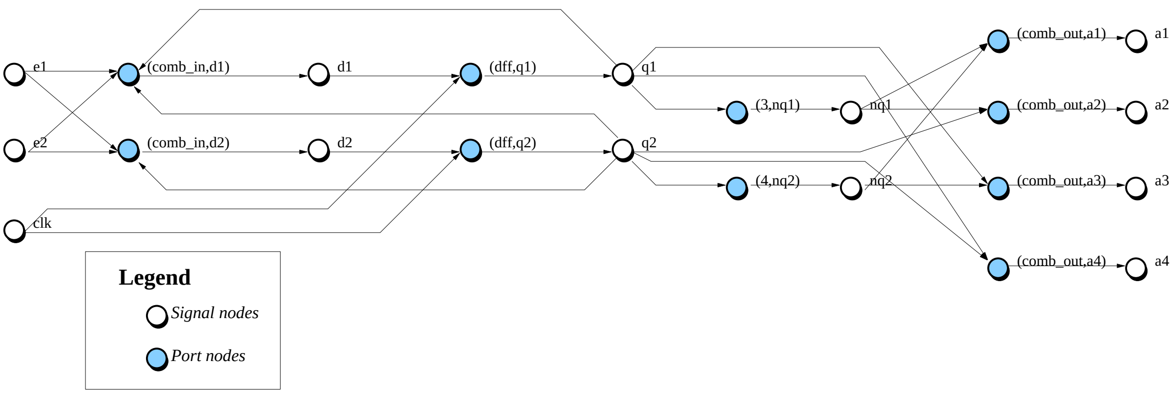
Each VHDL design is given by specifying the inputs and outputs using the VHDL entity declaration (called COUNTER in Figure 2), and the behavior using the VHDL architecture statements (called BEHAV in Figure 2). The behavior can be described using concurrent statements, e.g., processes or concurrent signal assignments. Individual processes consist of conventional sequential program code (sequential statements).

As our first model for VHDL programs, we chose a representation that used concurrent statements as diagnosis components and completely abstracted from individual values, referring to variables or signals merely as being correct or incorrect with regard to observations. This representation allows the debugging of very large programs with up to 10 MB of source code [1]. In a second model, we used expressions, sequential statements, and concurrent signal assignment statements as diagnosis components [7][11]. The behavior of the diagnosis components is given by the semantics of the original expression or statement.

The Dependency based model was developed during the *Design Diagnosis of VHDL specifications (DDV)* project funded by Siemens Austria as the first model used in the VHDLDIAG prototype. It uses functional dependencies between signals and variables. We say that a signal (variable) *x* depends on



(a) The program structure



(b) The functional graph representation

Figure 3 Two representations of the VHDL counter

other signals or variables *M* (written as *(x,M)*), if a value change of a signal (variable) $y \in M$ may cause a change of the value of *x*. Functional dependencies for the statements in the program can be easily computed from the source code. For example, consider line 18 of the COUNTER(BEHAV) program. The signal D1 depends on the signals E1 and E2 and the variable I2, i.e., the set of dependencies *{(D1,{E1,E2,I2})}* describes the assignment statement in line 18 in the dependency-based model.

Since we are interested in building a component model fragment for concurrent statements, we combine the dependency sets of the sequential statements. Consider again the COUNTER(BEHAV) program. The dependencies for the the process "comb_in" are given by *{(D1,{E1,E2,Q1,Q2}), (D2,{E1,E2,Q2})}*. Note that according to VHDL semantics, a signal *S* is not changed directly by a concurrent statement *C*. Instead *C* changes the value of the so-called driver of *S*, denoted by the pair *(C,S)*. All drivers of *S* ((C_i,S) for any component C_i that contains an assignment to *S*) influence the value of the signal *S* (with the choice between conflicting drivers being made by a special *resolution function* that decides on the right value to take). This special quality is represented by the functional dependency graph depicted in Figure 3. The dependencies of all concurrent statements intuitively induce a functional dependency graph by assuming that all signals are nodes and that two nodes are connected only if one signal depends on the other.

The semantics of a program's functional dependencies can be automatically encoded in logic. Consider a concurrent statement *C*. If *C* is correct and all its input signals have correct values, then the output signals of *C* also have to have correct values. The inputs and outputs coincide with the signals that occur in the functional dependencies of the statement. For example consider the concurrent signal assignment from line 29 in Figure 2 (which we will refer to as St_{29} from here on). The dependencies are given by $D_{29} = \{(NQ1,\{Q1\})\}$. The signal Q1 is an input for St_{29}, while NQ1 is an output signal. The model fragment produced for St_{29} is $\neg AB(St_{29}) \wedge ok(Q1) \rightarrow ok(St_{29},NQ1)$. The *AB* predicate (standing for „abnormal“) is used to indicate whether a statement is assumed to be correct or not. (It would be possible to use the *Ok* predicate for statements instead of $\neg AB$, but use of *AB* is more common in the model-based diagnosis literature.)

Apart from the model, we need the ability to specify observations which describe the actual behavior of the program. When using the dependency model, observations are simply classifications that tell us whether the value of signal *S* is correct or not, correctness being represented by the ground fact *ok(S)*, and and incorrect value for signals *S* by the fact $\neg ok(S)$. In addition, for all signals the system description contains the rule $ok(S) \wedge \neg ok(S) \rightarrow \perp$.

The VHDLDIAG model is actually somewhat more complex since cycles within the dependency graph require special handling. An initial model that propagated signal correctness using abstract time values resulted in *size* $O(|Signals|^2 \cdot |Concurrent\ Statements|^2)$ for the system description and prevented effective diagnosis of large systems. An improved model, described in [1], used direct modifications of the dependency graph to allow a system description of linear size.

The dependency model has been tested using real-world designs, with diagnoses being computed in a time frame suitable for the application. Even for larger designs, the diagnosis time was less than 10 minutes, which is satisfactory since simply executing the same design using a commercial simulator can already take hours to days (depending on the maximum simulation time limit). The quality of the diagnosis results varies from design to design depending on the properties of the functional dependency graph. However, no diagnoses are omitted. If too many diagnoses are computed, additional knowledge about the correctness or incorrectness of signal values has to be used. The current implementation provides assistance to the user by indicating which signals to examine next.

The exact model was developed as an extension to the abstract model to obtain better diagnosis results when debugging smaller programs. The mapping from VHDL programs to the logical representation forming the exact model proceeds in two stages. Initially, the program is converted into a component connection model, where components represent expressions and assignment statements, and connections represent signals and variables. This step is mainly concerned with programming language syntax issues, except for the handling of drivers and resolution functions, and distinguishing between variable and signal assignments. The second step involves the removal of cycles to reduce the computational complexity of diagnosis. Finally, we convert the resulting component connection model into a logical theory by using the components' behavior and the information about connectivity. The component behavior is given by logical sentences that express the VHDL semantics of the related program fragment.

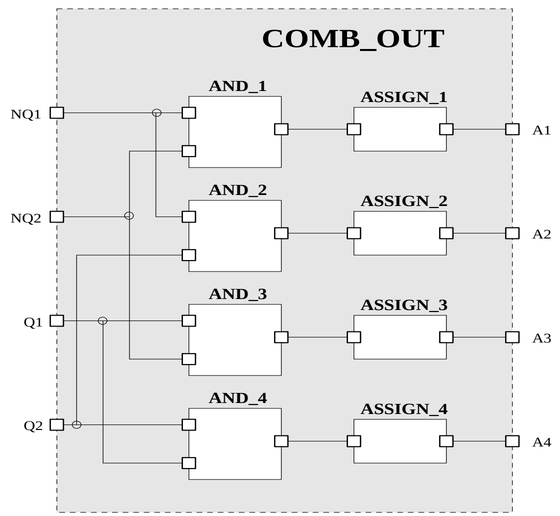


Figure 4 The converted comb_out process

Example: The process "comb_out" is mapped to the component-connection model given in Figure 4. The function **and** and all assignments are mapped to diagnosis components, e.g., AND_1 in the case of **and**. A similar conversion can be applied to the whole COUNTER(BEHAV) program. If we examine the functional dependency graph, we see that the signals Q1 and Q2 depend on themselves. The same cycle also occurs in the component-connection model. To avoid the use of temporal models, we can eliminate the cycle by splitting the signal in two parts. A separate signal representing one part, e.g., Q1, is used to represent the value at a time point $T\text{-}1$ and another signal, e.g., Q1_out, represents the value at time T. To compute diagnoses for time T we have to know the signal values at $T\text{-}1$. After removing all cycles, the logical model can be directly derived from the component-connection model. For example, the behavior of the AND component is given by $\mathbf{and}(A) \Rightarrow (\neg AB(A) \rightarrow out(A) = (in_1(A) \wedge in_2(A)))$. Observations for the detailed model are equations of the form $S = Val$, where S is a signal or variable and Val is a value.

Initial experiments using the exact model indeed show reductions in the sets of computed diagnoses. The overall number of diagnoses can be reduced further if other techniques such as those described in [8] and filtering criteria are applied. The models are described in more detail in [1][7][11].

Discussion

In this section we compare the dependency-based model to the value-based model according to their diagnostic capabilities using the following small VHDL example.

```
1.  entity DEMO is
2.    port(
3.      X,Y: in STD_LOGIC;
4.      U,V: out STD_LOGIC);
5.  end DEMO; [0.2em]

6.  architecture BEHAV of DEMO is
7.    signal I1,I2: STD_LOGIC := '0';
8.  begin
9.    I1 <= X;
10.   U <= not(I1);
11.   I2 <= I1 and Y;
12.   V <= I2;
13. end BEHAV;
```

In the DEMO(BEHAV) program the assignment statements in lines 9..12 are assumed to be executed in parallel. Now consider the specified behavior: If $X=0$ and $Y=1$, then the output of DEMO(BEHAV) must be $U=1$ and $V=1$. It can be easily proved that the program computes $U=1$ and $V=0$ contradicting the specification. Note that replacing statement 11 by "I2 <= I1 **or** Y;" is a solution to this debugging problem.

In the first step we use our dependency model for debugging. This model M^{FD}_{DEMO} comprises 4 components St_9, St_{10}, St_{11}, and St_{12}. The behavior is given by:

$$\neg AB(St_9) \wedge ok(X) \rightarrow ok(I1,9),$$
$$ok(I1,9) \rightarrow ok(I1),$$
$$\neg AB(St_{10}) \wedge ok(I1) \rightarrow ok(U,10),$$
$$ok(U,10) \rightarrow ok(U),$$
$$\neg AB(St_{11}) \wedge ok(I1) \wedge ok(Y) \rightarrow ok(I2,11),$$
$$ok(I2,11) \rightarrow ok(I2),$$
$$\neg AB(St_{12}) \wedge ok(I2) \rightarrow ok(V,12),$$
$$ok(V,12) \rightarrow ok(V)$$

Comparing the specified values with the evaluation results we obtain the following observations:

$$ok(X) \wedge ok(Y) \wedge ok(U) \wedge \neg ok(V).$$

Using this model and the definitions of model-based diagnosis [4] we get one conflict $\{ St_9, St_{11}, St_{12} \}$ and finally three single diagnoses $\{St_9\}$, $\{St_{11}\}$, *and* $\{St_{12}\}$. In this case, the result is equal to the result that would be obtained when using program slicing [9][10]. However, there are examples where the dependency-based approach improves on the program slicing results. A detailed discussion about this issue is outside the scope of this paper.

The graphical representation of the value-based model M^V_{DEMO} for the DEMO(BEHAV) program is depicted in Figure 5 and comprises 6 components: $ASSIGN_9$, $ASSIGN_{10}$, $ASSIGN_{11}$, and $ASSIGN_{12}$ representing the assignments and NOT_{10} and AND_{11} the functions used. Again we can compute conflicts using the component's behavior and structure given by the model and the observations. For this model the observations are equivalent to the specified values. In difference from M^{FD}_{DEMO} we now obtain two conflict sets: $\{ASSIGN_9, AND_{11}, ASSIGN_{11}, ASSIGN_{12}\}$ and $\{NOT_{10}, ASSIGN_{10}, AND_{11}, ASSIGN_{11}, ASSIGN_{12}\}$, and finally three single diagnoses: $\{AND_{11}\}$, $\{ASSIGN_{11}\}$, and $\{ASSIGN_{12}\}$. When comparing this result to the previous one we see that statement St_9 is not a single diagnosis when using a more sophisticated model. An explanation is as follows: Assume that St_9 ($\equiv ASSIGN_9$) is a single diagnosis. From $U=1$ and $\neg AB(ASSIGN_{10})$ and $\neg AB(NOT_{10})$ we obtain $I1=0$. Together with $\neg AB(ASSIGN_{11})$, $\neg AB(AND_{11})$, and $Y=1$ we derive $I2=0$. Using this fact and $\neg AB(ASSIGN_{12})$, we finally have $V=0$ contradicting the specification. Hence, $\{ASSIGN_9\}$ can not be a single diagnosis for model M^V_{DEMO}.

What we see using this example is that the more detailed model may lead to more focused results and therefore requires less user interaction for locating the bug. In addition, the value-based model can be extended to present to the user not only the bug location but also suggestions of how to correct the error. Moreover, when assuming structural correctness, i.e., no misused variables, the result for M^V_{DEMO} can be further improved. For example, assume diagnosis $\{ASSIGN_{12}\}$. We can compute the input for

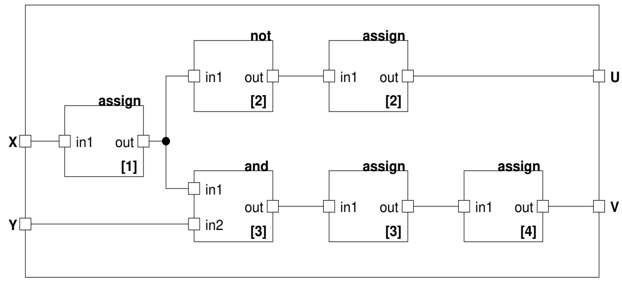


Figure 5 The converted DEMO(BEHAV) program

$ASSIGN_{12}$ and compare it to the expected output given by the specification.

X	***Y***	***in_1 ($ASSIGN_{12}$)***	***V(expected)***
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1

What we see is that (assuming structural correctness) the incorrect statement $ASSIGN_{12}$ can only be corrected by replacing it with another statement that also has one input and one output, but computes different output values for the same input value. This is not possible and therefore $\{ASSIGN_{12}\}$ is not a single diagnosis. With similar arguments we can eliminate $\{ASSIGN_{11}\}$ from the list of single diagnoses. Finally, only $\{AND_{11}\}$ remains! Although this result indicates the superior discrimination of the component-connection model, the dependency model is by no means useless. The most important reason is that debugging with the dependency model can be performed much faster than with the exact model. Therefore the dependency model is better suited to debugging large programs while the other model is for debugging smaller parts of a large program once identified as being buggy. In addition, there are program classes where both models behave equally good, e.g., programs implementing a chain of inverters. In this situation, where the value based model provides no additional discrimination, it is better to use the dependency model allowing to compute diagnoses much faster.

Conclusion

In this paper we have presented two different mappings from VHDL programs to logical models used for debugging. The principles behind these mappings can be applied to other programming languages. However, adaptions and extensions are necessary. Future research includes the development of models for recursive functions and loops, which are not necessary in the VHDL domain, since these constructs are not used by hardware designers.

Model-based reasoning provides a general theory also for program debugging regardless of programming paradigm. For example, the dependency model is similar to program slicing [10], while the exact model is related to Shapiro's algorithmic program debugging approach [5]. However, the model-based approach provides a higher degree of independence between a particular model and a particular diagnosis algorithm than the algorithmic debugging approach.

Acknowledgment

The work presented in this paper has been partially funded by Siemens Austria under research grant DDV GR 21/96106/4 and the Austrian Science Fund under Projects N Z29-INF and P12344-INF.

Address of Correspondence:

Markus Stumptner, Franz Wotawa:
Technische Universität Wien
Institut für Informationssysteme
Database and Artificial Intelligence Group
Favoritenstraße 9-11, A-1040 Vienna, Austria
Tel.: +43 (1) 58801 18433, +43 (1) 58801 18432
Email: {mst,wotawa}@dbai.tuwien.ac.at

Bibliography

[1] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. Artificial Intelligence, 111(2):3-39, July 1999.
[2] Ron I. Kuper. Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab, May 1989.
[3] William R. Murray. Automatic Program Debugging for Intelligent Tutoring Systems. Pitman Publishing, 1988.
[4] Raymond Reiter. A theory of diagnosis from first principles. Artificial Intelligence, 32(1):57-95, 1987.
[5] Ehud Shapiro. Algorithmic Program Debugging. MIT Press, Cambridge, Massachusetts, 1983.
[6] Markus Stumptner and Franz Wotawa. A model-based approach to software debugging. In Proc. of the International Workshop on Principles of Diagnosis (DX'96), Val Morin, Canada, 1996.
[7] Markus Stumptner and Franz Wotawa. VHDLDIAG+: Value-level Diagnosis of VHDL Programs. In Proc. of the International Workshop on Principles of Diagnosis (DX'98), Cape Cod, May 1998.
[8] Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In Proceedings 16th International Joint Conf. on Artificial Intelligence (IJCAI'99), Stockholm, Sweden, August 1999.
[9] Mark Weiser. Programmers use slices when debugging. Communications of the ACM, 25(7):446-452, July 1982.
[10] Mark Weiser. Program slicing. IEEE Transactions on Software Engineering, 10(4):352-357, July 1984.
[11] Franz Wotawa. Debugging synthesizeable VHDL Programs. In Proceedings of the Tenth International Workshop on Principles of Diagnosis (DX-99), Loch Awe, UK, 1999.

Markus Stumptneris an associate professor of computer science at the Vienna University of Technology, where he heads the model-based reasoning group at the Insitute for Information Systems. His research interests include configuration, model-based reasoning and software diagnosis, constraint satisfaction, and object-oriented techniques for software and data engineering. He received his MS and PhD in computer science from the Vienna University of Technology. He is a member of the IEEE Computer Society, ACM, and AAAI.

Franz Wotawa received a M.Sc. in Computer Science in 1994 and a PhD in 1996 both from Vienna University of Technology. He is currently an assistant professor at Vienna University of Technology and a member of the Database and Artificial Intelligence Group. His research interests include model-based and qualitative reasoning, configuration, planning, theorem proving and intelligent agents. Currently, Franz Wotawa works on applying model-based diagnosis to software debugging and has been involved in several projects in this domain. He is a member of IEEE Computer Society, ACM, and AAAI.