Using Multiple Models for Debugging VHDL Designs*

Franz Wotawa

Technische Universität Wien, Institut für Informationssysteme,
Database and Artificial Intelligence Group,
Favoritenstraße 9–11, A-1040 Vienna, Austria
wotawa@dbai.tuwien.ac.at

Abstract. Debugging is a time-consuming task especially for larger programs written by a group of people. In this paper we describe the use of multiple models for debugging VHDL designs, and presents some practical results. The models are derived from a general value-based model representing different fault situations that should be handled by a debugger. We propose the use of a probability-based selection strategy for selecting the most appropriate model in a given situation. For example large programs should be debugged using a model only distinguishing concurrent VHDL statements and not sequential statements. As a result of multi-model reasoning in this domain we expect performance gains allowing to debug larger designs in a reasonable time, and more expressive diagnosis results.

1 Introduction

Program debugging, i.e., locating and fixing a bug in a piece of software, is a difficult and time-consuming task. This holds especially for large programs written by a group of programmers. Because time to market is a major factor, tools for assisting a programmer in debugging are required. Traditional debuggers only allow to go step by step through a program and therefore require heavy user interaction. To avoid this problem several approaches for automating debugging has been proposed so far (see for example [15, 17, 18, 12, 3, 4, 11]). These approaches are more or less generally applicable, use different aspects of programs, e.g., dependencies between variables or fault probabilities, for debugging, and introduce their own algorithms. In contrast more recently it has been recognized that model-based diagnosis (MBD) can be adapted for software debugging [6, 1, 2, 8, 16]. In [6] the authors claim that MBD reduces the required user interaction when compared to [15]. The approach described in this paper makes use of previous MBD research in the software domain.

As in [8, 21, 20, 19] we concentrate on programs written in the hardware description language VHDL. VHDL is an ADA-like programming language and an IEEE standard [9]. A VHDL program consists of an entity declaration that describes the interface, and an architecture declaration that describes the behavior of the design of an digital circuit. The architecture declaration comprises processes which are working in parallel. Each process comprises sequential statements, e.g., assignments and conditionals. The communication between processes is handled using signals. Variables are

* The work described in this paper was partially supported by the Austrian Science Fund project P12344-INF and project N Z29-INF.

```
entity DECODER is
end DECODER;
architecture BEHAV of DECODER is
  signal ADDR, O, OI : BIT ;
  signal I1, I2 : BIT ;
begin
  . . .  -- Here should be VHDL code for test case
         stimulating signals ADDR, I1, I2
  work: process (ADDR) -- DECODER behavior
  begin
    if (ADDR = '0') then
      O <= I1;
      OI <= not(I1);
    end if;
    if (ADDR = '1') then
      O <= I2;
      OI <= not(I2);
    end if;
  end process work;
end BEHAV;
```

Fig. 1. VHDL DECODER(BEHAV) program

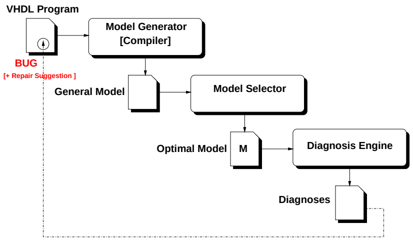

Fig. 2. Structure of an intelligent debugger

only allowed in processes. For more details about VHDL have a look at [13]. Figure 1 shows a small VHDL program implementing a decoder. VHDL is a general language, allowing to specify (possible recursive) functions and data types. However, most programs are element of the register transfer level (RTL) subset where some syntactically restrictions apply. The reason for using this subset is that RTL programs can be (semi-)automatically compiled into a gate level representation from which there is only a small step to real hardware. Several syntactical restrictions apply to RTL programs, e.g., loop statements are only allowed if the number of loops is known at compile time, delay times must not be specified, and others. More informations about the RTL subset can be found in [10].

The basic idea about the application of MBD in software debugging is to translate a program into its logical representation, and use this representations together with a diagnosis engine [14, 7] to compute diagnoses. The resulting diagnoses can be mapped back to their associated program fragments afterwards. In [8] and [21] VHDL concurrent statements, i.e., processes, are considered as diagnosis components, and signals as connections between components. This representation allows to state that a process may has a bug or not. The models introduced in [20, 19] ensure more detailed results. This is done by mapping (sequential) statements and expressions to components and signals (and variables) to their connections. Although, this model is similar to the one we describe in this paper, it is not as general as ours and causes problems in some situations because of the underlying assumptions, e.g., structural correctness of the model.

Our approach of debugging RTL programs introduces a general model for VHDL programs with the capability of locating all functional faults. We define functional faults as faults occurring in a compilable program causing an unexpected behavior for at least one test case. Functional faults can be corrected by replacing expressions, adding or removing lines of code, or by changing variables or signals. The potential search space for locating and correcting functional faults is infinite. Therefore, we assume that the corrected program is a small variant of the original one. Although, this assumption restricts the search space, debugging is still time consuming especially for large programs, leading to a system description SD, i.e., a model,comprising a large amount of diagnosis components $COMP$. Using MBD for computing all single diagnosis has a worst case time complexity $O(|COMP|^2)$ assuming that SD is a propositional horn clause formula. Since, our extended model has more components as the previous model from [20, 19], debugging time increases. To avoid this problem we derive smaller models from the general one. A debugger then selects an appropriate derived model using knowledge about the current situation, e.g., source code size, type of the misbehavior, and others. In the next step this derived model is used for debugging instead of the general one.

The introduced debugger concept depicted in Figure 2 consists of a compiler, a model selector, and a diagnosis engine. The compiler transforms a VHDL program into a general model. The model selector takes the general model and uses additional knowledge, e.g., probability distributions or file size, to derive the diagnosis model. This is used (together with the expected behavior) by the diagnosis engine to compute diagnoses, which are mapped back to the source code. An advantage of this concept is that the used model is adapted to the current situation and overall diagnosis time is reduced. Note, that the situation can change during debugging, because of additional knowledge, e.g., expected values of signals provided by the user. Hence, the diagnosis model may change during debugging. However, in the worst case the overall diagnosis time is of the same order as the diagnosis time of the general model, but we expect a better performance in the average case.

The paper is organized as follows. In the next section we give a short overview of MBD and introduce the model. Afterwards, we show how smaller models can be derived, and how a debugger can handle multiple models. Some preliminary results obtained by using such a derived model are given, indicating that MBD substantially help finding bugs in hardware designs. Finally, we conclude the paper and discuss future research issues.

2 The general model

Model-based diagnosis [14, 7] is characterized by using a model SD, that describes the structure and the behavior of a system, and a set of observations OBS to determine a set of diagnoses, i.e., components from $COMP$ responsible for a misbehavior. Formally, a diagnosis problem is a triple $(SD, COMP, OBS)$. A component set $\Delta \subseteq COMP$ is a diagnosis iff $SD \cup OBS \cup \{\neg AB(C) | C \in COMP \setminus \Delta\} \cup \{AB(C) | C \in \Delta\}$ is consistent. A diagnosis is minimal if no proper subset is a diagnosis. The predicate $AB(C)$ ($\neg AB(C)$) denotes that component C behaves wrong (correct). This basic definition can be extended to handle fault modes, i.e., component modes associated with a specific behavior, e.g., stuck-at-0 in the digital domain. For our purposes we assume that each component C has modes $\Gamma(C)$, denoted by symbols starting with an uppercase letter, e.g., S0 for stuck-at-0, whereas the mode AB denotes the situation where the faulty behavior is not known. In the extended diagnosis definition a diagnosis is an assignment of modes to components leading to a consistent theory. Important to MBD is the existence of a component-oriented model. Such a model exists for systems from technical domains and the software domain.

The general VHDL model M_G of a program Π reflects the semantics of VHDL and the syntactical properties of Π. As stated previously M_G is automatically generated from Π. This transformation maps statements and expressions to diagnosis components and signals (and variables) to connections. A VHDL program Π is converted by successively converting the processes declared by the VHDL architecture declaration. Signals used in an expression or the sensitivity list of a process, e.g., signal *ADDR* from process *work* in Figure 1, are inputs, and signals used as target of an assignment statement, are outputs of the process and its associated diagnosis component. Note, that it is possible that several processes have the same signal as an output. In this case VHDL introduces a resolution function that allows to compute a single value for the signal. For the general model M_G in this case a diagnosis component representing the resolution function is used.

The conversion of a single process is done by introducing a diagnosis component for the process and successively converting each sequential statement starting from the first and going to the last. Again, each signal or variable used in an expression occurring in a sequential statement is an input, and each statement or variable used as target of an assignment are outputs of the statement. In order to reflect the semantics of VHDL there is a difference between connections representing variables and signals. For every output variable a new connection is created, that is connected with an input of a statement converted later. Instead, every input signal for a statement is connected using the same connection. The connection created for every signal is unique. The different handling of variables and signals used in an expression correctly represents the VHDL semantics. In VHDL the signal value is changed only after leaving a process. More details and a formal definition of the conversion process (returning a more simplified model) can be found in [20, 19]. Note, that resulting structural model may contain cycles. They must be eliminated by introducing additional connections associated with signals (see [20, 19]). This procedure is similar to the conversion of sequential circuits to combinational circuits [5]. During the rest of this paper, we assume that the model M_G is acyclic. In the following we describe the created diagnosis components for each considered statement.

- **Processes**: According to the VHDL semantics processes must be executed if at least one signal S from the sensitivity list has changed its value. In this case the implicit available signal $S'EVENT$ is true. To represent this behavior correctly, we introduce a new diagnosis component $PROC(C)$ for each process. This component has a port sen_S for every signal S from the sensitivity list, and ports ex_S, nex_S, and out_S for every output signal of the process. The ex_S ports are connected to the last signal connection for S obtained during converting the sequential statements. nex_S is connected to the original signal connection. This connection represent the state of S before executing a process. The output out_S represents the state of S after considering the process. The sen_S ports are connected to the $S'EVENT$ connections. The behavior of a process is simple. If at least one sen_S port has the value true, then the value of ex_S must be equal to the value of out_S. Otherwise, the value of nex_S is equal to out_S.
  $\neg AB(C) \Rightarrow (\exists_S sen_S(C) = true \rightarrow \forall_X ex_X(C) = out_X(C)$
  $\neg AB(C) \Rightarrow (\forall_S sen_S(C) = false \rightarrow \forall_X nex_X(C) = out_X(C)$
  The diagnosis components for processes have only two modes $\{\neg AB, AB\}$.

- **Signal assignments** are mapped to a diagnosis component $SA(C)$ having a port in and a port out. If the assignment is correct, i.e., use the correct target signal and expression, the input port has the same value as the output port.
  $\neg AB(C) \Rightarrow (in(C) = out(C))$
- **Conditionals** are mapped to a diagnosis component $COND(C)$ with a port $cond$ connected with the converted conditional expression, and ports $then_S$, $else_S$, and out_S connected with every connection associated with a signal or variable S used as target in an assignment statement either in the then-branch or the else-branch of the conditional. The behavior is defined as follows:
  $\neg AB(C) \Rightarrow (cond(C) = true \rightarrow \forall_S then_S(C) = out_S(C))$
  $\neg AB(C) \Rightarrow (cond(C) = false \rightarrow \forall_S else_S(C) = out_S(C))$
  In addition we introduce the fault mode $WCOND$ that indicates a bug in the conditional expression.
  $WCO(C) \Rightarrow (cond(C) = false \rightarrow \forall_S then_S(C) = out_S(C))$
  $WCO(C) \Rightarrow (cond(C) = true \rightarrow \forall_S else_S(C) = out_S(C))$
  Hence, a conditional has three modes $\{\neg AB, AB, WCOND\}$. Note that the considered VHDL subset allows neither to define a general loop nor a recursive function. Therefore, the $WCOND$ mode never causes an infinite loop (which can be the case if the restrictions do not apply).
- **Expressions** consists out of signal or variable accesses, constants, and functions. For each category we use a different mapping.
  - **Signal access** are mapped to a diagnosis component $SAC(C)$, with one output out, and a port in_S for every signal S declared in the program. The accessed signal T is stored in a predicate $sig(C, T)$. The correct behavior of the diagnosis component is as follows:
    $\neg AB(C) \Rightarrow (sig(C, T) \rightarrow out(C) = in_T(C))$
    In addition for every signal $S \neq T$ we add the known misbehavior:
    $WSIG_S(C) \Rightarrow (out(C) = in_S(C))$
    to the system description. This part of the model allows the diagnosis engine to switch between available signals and to explain a wrong signal access.
  - **Constants** are mapped to a diagnosis component $CONS(C)$ with one output out delivering the constants value stored in the predicate $const(C, V)$.
    $\neg AB(C) \Rightarrow (const(C, V) \rightarrow out(C) = V)$
    Additionally, we introduce fault modes delivering other constant values depending on the type of the constant. For boolean constants for example we add the following rule:
    $WCON(C) \Rightarrow (const(C, V) \rightarrow out(C) = \neg V)$
    For other domains similar rules can be formalized. However, for describing digital circuits the considered domains must be finite.
  - **Functions** $f \in \{and, or, not, \ldots\}$ are mapped to corresponding diagnosis components $f(C)$, e.g., for an **and** function we have a component $AND(C)$ with ports $in1, in2, out$ and the following behavior:
    $\neg AB(C) \Rightarrow (out(C) = in1(C) \wedge in2(C))$
    In addition we assume fault models that represent repair suggestions, e.g., a wrong **and** function that should be repaired by replacing it with an **or** function.

$$WFUN_{or}(C) \Rightarrow (out(C) = in1(C) \vee in2(C))$$
$$WFUN_{nand}(C) \Rightarrow (out(C) = \neg(in1(C) \wedge in2(C)))$$
$$\ldots$$

For space reasons the rules for converting variables used in assignments or expressions is not included. They are similar to the one given for signals, with a slightly

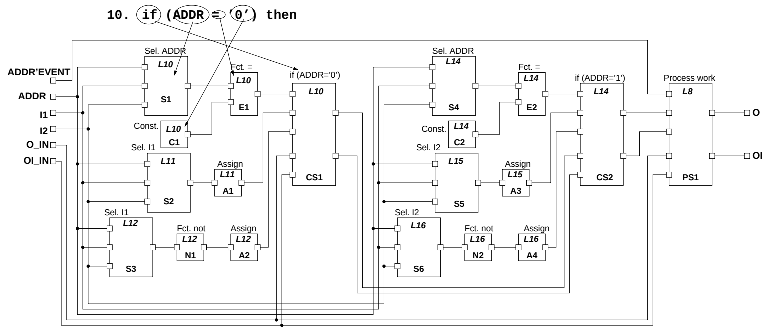

Fig. 3. A model of the DECODER(BEHAV) program

different handling of connections accordingly to the semantical differences between variables and signals in VHDL. Moreover, the computed model is not intended to handle faults related to wrong target signals. Handling this kind of faults would makes the resulting model more complicated but would has no influence on further contributions of this paper, i.e., the derivation of less general models and the handling of multiple models for optimizing the debugging process using MBD. Using the above conversion our running example from figure 1 is converted to a model depicted in Figure 3. The model comprises 19 diagnosis components: $\{S1, S2, S3, S4, S5, S6\}$ (Signal access), $\{A1, A2, A3, A4\}$ (Assignments), $\{N1, N2\}$ (Negations), $\{E1, E2\}$ (Equivalence functions), $\{C1, C2\}$ (Constants), $\{CS1, CS2\}$ (Conditionals), and $\{PS1\}$ (Process *work*).

The general model M_G of a program Π consists of a system description SD_G and a set of components $COMP_G$. SD_G can be divided into a structural part SD_{GS} and a behavioral part SD_{GB}. The structural part comprises knowledge about the interconnection an corresponds with the syntactical structure of Π while SD_{GB} is only defined once for a programming language and used in all general models. Only some instantiation informations, e.g., $const('1', C2)$ in our example, which is assumed to be part of SD_{GB} has to be added to get the final model.

3 Deriving models

With the introduced general model M_G we can handle almost all fault situations, including the wrong use of signals or variables, wrong functions, wrong conditions, and others. Therefore, in some cases we get too many different diagnoses. Moreover, they are difficult to handle by a programmer. Consider for example that the diagnosis engine delivers a diagnosis corresponding to a structural bug[1] and another (different) diagnosis corresponding to a wrong function. Without further guidance it is difficult (1) to see the different nature of the bugs, and (2) to distinguish between the diagnoses. Therefore, we suggest the use of different models handling different kind of bugs, e.g., structural

[1] A structural bug is due to a wrong use of a signal or variable while a functional bug is due to a wrong use of a function or operator.

bugs and wrong functions. Another argument motivating the use of multiple models is the performance of diagnosis. A sub-model of M_G with less diagnosis components reduces the time for computing all diagnoses in the average. In order to allow MBD for debugging larger designs, it is an requirement to use the most appropriate model for computing bug locations in a reasonable time.

For our purposes we distinguish two types of sub-models for $M_G = (SD_G, COMP_G)$:

- **Partial models** are models derived from a general model M_G, where the system description is extended using correctness assumptions for the non-empty set of components $CO \subseteq COMP$, and the set of components is reduced to $COMP = COMP_G \setminus CO$. Formally, a partial model is a tuple $(SD \cup COA, COMP)$ with $COMP = COMP_G \setminus CO$, and COA set of assumptions about the state of the components in CO, e.g., correctness of a component $\neg AB(PS1)$.
- **Hierarchical models** are models derived from M_G where new components representing a set of components from $COMP_G$ are introduced, together with rules specifying the correct behavior of the new components. $(SD \cup MAP, COMP)$ with $COMP \cap COMP_G = \emptyset$, and MAP is a set of rules defining the correct behavior of a component $C \in COMP$ using modes of components $C' \in COMP_G$, e.g., $\neg AB(work) \rightarrow (\neg AB(PS1) \wedge \neg AB(CS1) \wedge \ldots)$.

Because of the definition both sub-models have less components than the original model M_G. The size of the system description slightly increases but should have almost no effect on the overall run-time. Hence, if it is possible to select the right sub-model for debugging, we expect substantial savings of the overall diagnosis time.

The following propositions follows directly from the definition of model-based diagnosis and partial models.

Lemma 1. *A diagnosis for a partial model $M = (SD \cup COA, COMP)$ and observations OBS derived from a general model M_G exists iff $SD \cup COA \cup OBS$ is consistent.*

Hence, a minimal requirement for partial model is that $SD \cup COA$ is consistent. For the VHDL debugging domain we distinguish the following models:

- *Correct structure* M_{COR} For this partial model we assume that the structure is correct, i.e., we set all signal access components and process components to the correct state. In our running example we have $CO = \{S1, S2, S3, S4, S5, S6, PS1\}$. This assumptions allows us to use multiple test cases and filter criteria as described in [19, 16].
- *Structural faults* M_{SF} In the structural fault model we assume that only signal access components can be faulty, i.e., all other components are assumed to behave correct. For our running example we get $CO = \{E1, E2, A1, A2, A3, A3, N1, N2, CS1, CS2, PS1\}$ and only the statements $S1, S2, S3, S4, S5, S6$ remain as diagnosis components.
- *Wrong condition* M_{WCOND} This partial model is used to locate faults in the condition part of a conditional statement. It is required that the observations include

the expected value of the condition expression. All components are set to the correct mode except the component associated with the conditional statement having a bug in the condition, i.e., the conditional is set to WCO. The set of diagnosis components for M_{WCOND} comprises all components that are associated with the condition expression (and its sub-expressions). M_{WCOND} should only be used if a wrong condition has been located using another model, e.g., M_{COR}.

– *Concurrent statements* M_{CONC} This model is a hierarchical model considering processes as single diagnosis components. In our running example all components are mapped to a single diagnosis component representing the process. M_{CONC} is intended to be used for larger programs in order to reduce the number of diagnosis components and therefore to reduce the overall diagnosis time.

This four derived models represent different aspects of debugging VHDL programs. Similar models are expected to be useful for debugging programs written in other programming languages. In order to reduce the overall debugging time the selection of the best model in a current situation is a crucial requirement. We define the best model informally as the model that lead to the localization of a bug in a minimum number of required questions about values at specific points in the program and in the shortest time. For example if the program is large, then the hierarchical model should be used. After identifying one process a more detailed model can be used for further discrimination. A more detailed view on the model selection process has been left for future research. But we expect that a probability-based technique similar to [3, 4] would be a good choice in this context.

4 Results

In this section we give first results for model M_{COR} using multiple test cases [16] and filtering as described in [19]. We show that both techniques significantly reduce the number of computed candidates. The empirical results based on three different VHDL designs with several faulty variants. The faulty designs where derived from the correct programs by introducing bugs like changing constant values or functions. Table 1 gives the programs together with the number of different architectures and the range of computed diagnosis components for each designs. Design COUNTER1 implements a simple digital counter. COUNTER2 implements a counter with a 1 from 4 output. The ENCODER1 and ENCODER2 program implement two variants of a BCD to 7-segment LCD display decoder.

The results of diagnosis using our prototype implementation are given in Table 2. In this Table the column $DIAG$ gives the number of candidates obtained by building the union of all diagnoses computed for all test cases without applying the filtering rules, i.e., $DIAG = | \cup_{\Delta \in DIAGNOSES_i} \Delta|$. $DIAG\_MT$ gives the number of diagnosis for the multiple test case diagnosis where the intersection of diagnoses is used, again without filtering. In column $DIAG\_MTF$ the resulting number of diagnoses after applying multiple test case diagnosis and filtering is given. Column EXP is true, whenever the expected diagnosis remains after multiple test case diagnosis and filtering was applied. Note, that the expected diagnosis was always computed whenever neither

Entity	Num. Arch.	Diag. Components
COUNTER1	11	23…24
COUNTER2	6	41…42
ENCODER1	5	440
ENCODER2	5	440

Table 1. VHDL Designs

Design	DIAG	DIAG_MT	DIAG_MTF	EXP
COUNTER1(BEHAV1)	12	6	1	yes
COUNTER1(BEHAV2)	6	4	1	yes
COUNTER1(BEHAV3)	11	7	1	yes
COUNTER1(BEHAV4)	13	11	3	yes
COUNTER1(BEHAV5)	14	7	1	yes
COUNTER1(BEHAV6)	14	9	2	yes
COUNTER1(BEHAV7)	8	5	5	no
COUNTER1(BEHAV8)	16	11	3	no
COUNTER1(BEHAV9)	17	2	2	no
COUNTER1(BEHAV10)	8	6	5	no
COUNTER2(BEHAV1)	5	1	1	yes
COUNTER2(BEHAV2)	2	0	0	no
COUNTER2(BEHAV3)	4	3	2	no
COUNTER2(BEHAV4)	11	3	1	yes
COUNTER2(BEHAV5)	10	7	3	yes
ENCODER1(BEHAV1)	79	3	1	yes
ENCODER1(BEHAV2)	85	8	5	yes
ENCODER1(BEHAV3)	60	59	2	yes
ENCODER1(BEHAV4)	15	15	1	yes
ENCODER2(BEHAV1)	65	3	1	yes
ENCODER2(BEHAV2)	67	9	6	yes
ENCODER2(BEHAV3)	55	55	1	yes
ENCODER2(BEHAV4)	63	24	1	yes

Table 2. Results of the multiple test case diagnosis

filtering nor multiple test case diagnosis was applied. It is worth noting that $DIAG$ gives an upper-bound of the number of minimal diagnoses while $DIAG\_MTF$ can be viewed as lower-bound.

The obtained result indicates the use of multiple test cases together with the application of filter rules indeed drastically reduces the number of computed diagnoses. In 11 out of 17 cases, where the expected result was computed, our approach delivers only 1 diagnosis. In the other cases at most 6 candidates remain. However, there are cases, where our approach computes wrong diagnoses. This happens only when structural faults causes the wrong behavior. Since, M_{COR} together with multiple test case diagnosis is not allowed to be used in case of structural faults, the result is the expected one.

5 Conclusion

This paper describes how multiple models help to decrease debugging time of VHDL designs. The multiple models are derived from a general model handling almost all fault situations. We propose the selection of a most probable model in the given situation. A situation is characterized by propositions. For example the proposition *large program* is valid if the program exceeds a specific size, say 10 k-Byte.In this case a hierarchical modeling approach is expected to be used in order to allow for computing diagnoses in a short time. Future research should focus on the model selection process. Probabilities of models with respect to a given situation should be computed. To do this a concise analysis of the probability of bugs and their effects is needed. In addition to the analysis of the model selection process, models for a greater subset of VHDL (or better full VHDL) are of interest. However, the contributions of this paper, i.e., the general model, the computation of derived models, and the view of debugging as multi-model reasoning process proved a good foundation for future research in the domain of intelligent debugging.

References

1. G. W. Bond and B. Pagurek. A Critical Analysis of "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs". Technical Report SCE-94-15, Carleton University, Dept. of Systems and Computer Engineering, Ottawa, Canada, 1994.
2. Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis*. PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Canada, 1994.
3. Lisa Burnell and Eric Horvitz. A Synthesis of Logical and Probabilistic Reasoning for Program Understanding and Debugging. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence*, pages 285 –291, 1993.
4. Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, pages 31 – 41, 1995.
5. Kwang-Ting Cheng and Angela Krstic. Current Directions in Automatic Test-Pattern Generation. *IEEE-Computer*, 32(11), November 1999.
6. Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*, Chambery, August 1993.
7. Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
8. Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.
9. IEEE. IEEE Standard VHDL Language Reference Manual LRM Std 1076-1987, 1988. Institute of Electrical and Electronics Engineers, Inc. IEEE.
10. IEEE. IEEE P1076.6/D1.12 Draft Standard For VHDL Register Transfer Level Synthesis, 1998. Institute of Electrical and Electronics Engineers, Inc. IEEE.
11. Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.
12. Bogdan Korel. PELAS–Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, 14(9):1253–1260, 1988.
13. Zainalabedin Navabi. *VHDL Analysis and Modeling of Digital Systems*. McGraw-Hill, 1993.
14. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1), 1987.
15. Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.
16. Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*, Stockholm, Sweden, August 1999.
17. Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.
18. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
19. Franz Wotawa. Debugging synthesizeable VHDL Programs. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, 1999.
20. Franz Wotawa. New Directions in Debugging Hardware Designs. In *Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 1999. Springer, Lecture Notes in Artificial Intelligence 462 (LNAI 462).
21. Franz Wotawa. Debugging VHDL Designs using Model-Based Reasoning. *Artificial Intelligence in Engineering*, 14(4):331–351, 2000.