Dissertation

Applying Model-Based Diagnosis to Software Debugging of Concurrent and Sequential Imperative Programming Languages

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors
der technischen Wissenschaften

eingereicht an der Technischen Universität Wien

Technisch Naturwissenschaftliche Fakultät

von

Franz Robert Wotawa, A-1140 Wien, Wiengasse 6/7/3, Österreich
16.12.1968, Wien

Wien, im Mai 1996

Kurzfassung

Diese Dissertation beschäftigt sich mit der Anwendung modellbasierter Diagnosetechniken auf das Debuggen von imperativen Programmen. Zu diesem Zweck werden Programme in Diagnosesysteme umgewandelt. Diese Umsetzung kann prinzipiell auf mehrere Arten geschehen und beeinflußt die Aussagekapazität des Diagnosesystems bezüglich möglicher Programmfehler. Drei verschiedene Konvertierungen werden im Rahmen dieser Dissertation beschrieben und miteinander in Verbindung gesetzt. Eine Evaluierung bezüglich der Diagnosekapazität, der Fehler sowie der Anwendbarkeit auf große Programme ist ebenfalls Teil der vorliegenden Arbeit.

Das erste Modell verwendet die Abhängigkeiten zwischen Signal und Variablen, wie sie in Programmen vorkommen, zur Diagnose. Dieses einfache Modell ermoeglicht selbst die Diagnose von sehr großen Programmen (Source-Code-Größe bis zu 10 MB). Die Abstraktion führt jedoch dazu, daß die Güte der Diagnose gemessen an der Diagnosekandidatenanzahl im Vergleich zur Gesamtanzahl von Diagnosekomponenten von der Programmstruktur wesentlich abhängt. Um dieses Manko zu beheben, wurden die beiden anderen Modelle entwickelt.

Im zweiten Modell werden Evaluationsgraphen in Diagnosesysteme umgewandelt. Durch das Einbinden zusätzlicher Semantikinformation kann die Güte der Diagnose stark verbessert werden. Die Gesamtperformanz bezüglich der Fehlerberechnung sinkt jedoch, sodaß nur kleinere bis mittlere Programme diagnostiziert werden können. Nachteilig wirkt sich auch die Tatsache aus, daß zwar Fehler gefunden, Reparaturvorschläge jedoch nicht gemacht werden.

Das dritte Modell erlaubt die Berechnung von Reparaturvorschlägen basierend auf Ausdruckersetzungen. Diese Ersetzungen werden unter der Annahme durchgeführt, daß das richtige Programm und das falsche Programm sehr ähnlich sind, d.h. nur wenige Änderungen notwendig sind, um ein nicht korrektes Programm in ein korrektes umzuwandeln.

Alle drei Modelle können miteinander integriert werden. So kann das Resultat des ersten Modells zur Fokusierung bei der Diagnoseberechnung mit dem zweiten Modell verwendet werden und dieses mit dem dritten. Auf diese Art wurde gezeigt, daß die Gesamtperformanz gesteigert werden kann.

Der vorgestellte Diagnoseansatz zum Software-Debugging kann somit sowohl zur Fehlerlokalisierung als auch in einem eingeschränktem Maß zur Fehlerbehebung eingesetzt werden. Es werden sowohl Fehler, die auf falsch verwendete Programmzeilen oder Ausdrücke, als auch Fehler, die auf fehlenden Programmteilen beruhen, gefunden. Typfehler können ebenso lokalisiert werden. Der in dieser Dissertation beschriebene Ansatz kann mit einem Verifikationssystem und bestehenden Programmentwicklungsmethoden verbunden werden.

Abstract

This thesis describes the application of model-based diagnosis to software debugging of imperative languages. To do this, programs are converted into diagnosis systems. Three different conversions are presented which have different diagnosis capabilities. A comparison of the three models, which are the results of the conversion functions, is part of this thesis.

The first model uses functional dependencies between signals and variables for diagnosis, allowing to diagnose very large programs of about 10 MB source code size. A disadvantage is that the number of diagnoses highly depends on the program structure and the knowledge of incorrect signal and variable values. To avoid these problems two other models have been investigated.

The second model uses evaluation graphs which are directly converted to diagnosis systems. The introduction of program semantics to diagnosis improves diagnosis results but increases the overall computation time, such that it usage is restricted to small and medium programs. Repair proposals can not be provided by the second model.

In the third model disadvantages of the two other models are avoided. This is done by introducing expression replacements as fault models for diagnosis. The underlying assumption is that the correct program is only a small variant of the original (wrong) program. Fixing a bug is then reduced to replace an expression or statement by a small variant. The third model can only be used for diagnosis of small programs.

A big advantage of the chosen model-based approach is that the presented models can be combined to improve the overall diagnosis process. For example, the diagnoses delivered by the first model can be used as focus for the second model. On of the results of this thesis is that the performance can be substantially improved.

The program debugging approach presented here can be used for fault localization and repair. Faults related to faulty statements and expressions, and also related missing program parts can be found. Type errors can be handled as well. In addition, the diagnosis approach can be easily combined with verification systems and current program development tools.

Contents

To my parents Eveline and Franz,
and my wife Michaela
for love and support, and
my little son Franz Wolfgang
for being here

Foreword

This dissertation describes how programs can be diagnosed by introducing formal systems which are related to programs. Every program can be viewed as world and the associated formal system as representation of the world. This association is similar to the one presented 1921 by Ludwig Wittgenstein in his tractatus logic-philosophicus. In this work he assumed that the whole world can be described by formal sentences, where new knowledge about the world can be derived from. Questions which cannot expressed within this system can also not be answered. Although he critically revised his view of the world in his later work, I think the basics behind still remain the same.

Similar to modeling the world, the problem of modeling a formal system which can be used for diagnosis and is related to syntax and semantics of a program, does not have a single solution. Various assumptions about what is to be expressed and how is it to be expressed. The resulting model has an influence to the diagnosis capability and results delivered to the user of such as system. Because of the similarities I have chosen the leading paragraphs of the tractatus as quotations at the beginning of each chapter. Please note that there is not a one to one relation between the quotations and the content of a chapter, but there are connections between the tractatus and the presented work.

Vienna, 1996

Franz Wotawa

Chapter 1

Introduction

1. Die Welt ist alles, was der Fall ist.

Ludwig Wittgenstein, 1921
Tractatus logico-philosophicus
Logisch-philosophische Abhandlung

This dissertation presents basics in applying model-based diagnosis MBD to the software domain. While automatic software debugging is not new [Sha83], traditionally little effort has been spent in applying model-based techniques to this domain. Only [CFD93] and subsequently [BP94, Bon94] have previously used MBD for debugging of logic programs. However, most of todays software is written in imperative programming languages such as C, C++, or VHDL. We therefore concentrate on basic principles of using MBD in this domain. This idea was the basis of the *Design Diagnosis of VHDL Specifications* (DDV) project sponsored by Siemens Austria during 1994 to 1995. The aim of the project was to show how very large VHDL programs could be debugged. VHDL is a concurrent programming language used to design electronic devices such as ICs and ASICs. In this thesis some of the results of this project are presented. See also [FSW95].

Because of the aim of diagnosing large programs the language model used was very abstract, providing a good performance but lack in giving explanations for bugs and how to correct them. The focusing on relevant diagnosis capability should be also improved. To avoid these drawbacks, new more concise programming language models must be developed. Two possible ways of doing this are also presented in this dissertation. The first uses the computation trace while the second one uses possible replacements to improve diagnosis capability. A comparison of both approaches is given.

The presented basic ideas, language models, and questions of on how further research should be based, should allow to build automatic debugging programs in the near future, which can be used also for very large programs. However the presented approach is currently limited to the diagnosis of functional faults. Missing parts of programs, e.g., statements and expressions, can be found, too. Non-termination errors, type errors and syntax errors are not considered.

The thesis is organized as follows. Section 1.3 gives an overview of all activities related to automatic programming. It is followed by a short introduction to the domain of software debugging. Chapter 1.2 presents basic definitions, theorems and algorithm used in this thesis. The first section is related to model-based diagnosis while the second one is related to programming languages. Questions of what is a programming language, what are computation traces, replacements and so on, are answered in this part. Chapter 2 gives an overview of syntax and semantics of a subset of VHDL. The semantics is described denotationally and used while building behavioral modes for diagnosis. Examples should clarify the given definitions.

After the basics we present the system model used in the DDV project in chapter 3. A motivation of why abstract models should be used for diagnosis of very large programs, an introduction into functional dependencies and how to use them, and various improvements allowing it to use the model in practice are presented. A section on implementation results, performance and usability is given, too. This abstract model is followed by a description of how execution traces (chapter 4) and fault models (chapter 5) can be used for diagnosis. A chapter comparing all three approaches concludes this piece of work. Finally, future improvements and directions are given.

1.1 Basic Principles and Ideas of Software Debugging

We present the underlying ideas regarding software debugging by using a simple VHDL example. Consider the following program, implementing a simple RAM cell.

Example 1:

```
entity MEM is
     port (
          signal clk, de, rw :  in std_logic;
          signal data :  inout std_logic_vector(1 to 4) );
end MEM;
architecture BEHAV of MEM is
begin
     MEM_UNIT: process
          variable mem:  std_logic_vector(1 to 4)
     begin
          if clk = '1' and de = '1' then
               if rw = '1' then
                    mem := data;
               else
                    data <= mem;
               end if;
          end if;
          wait on clk;
     end process;
end BEHAV;
```

This program makes use of VHDL *entity* and *architecture* declarations. Entities define components of a VHDL program and describe their interface. Architectures define the behavior of entities. Different architectures can be associated with different instances of the same entity. Architectures consists of concurrent statements such as the above `process` statement. Statements used inside a process are executed in sequence. The explanation of the program above is as follows: On every rising cycle of the clock `clk`, when data enable `de` is high, read (`rw = '0'`) or write cycles (`rw = '1'`) can be performed. The process then waits until an event on `clk` occurs.

□

Assume further that the above program has been changed by introducing an encryption and decryption function leading to a modified `BEHAV_2` architecture.

```
...
     mem := encrypt(data);
else
     data <= decrypt(mem);
end if;
...
```

Suppose that instead of using the correct `decrypt_new` function, the wrong `decrypt` function has been used, resulting in incorrect behavior. Note that the incorrect behavior is usually detected by applying test cases. A test case specifies the input and the expected output. We say that a program passes a test case iff using the given input leads to an output of the program which is equal to the expected one. Detecting an incorrect behavior naturally raises two questions:

1. Where is the related bug located within the program?
2. How can we correct the bug without introducing a new one?

Consider for example the above VHDL program. If we detect that the signal data at some time (e.g., 10000ns) has the bit sequence "0010" instead of "0011" assigned, the answers to both questions are of interest. An answer to the first question can be obtained by tracing back from the assignment to the point where the program behaves correctly. Using the given behavior of the program parts can be used to focus only on those parts which may cause the misbehavior. Once a bug has been detected it can be corrected. This correction can be done by successively replacing the bug by program parts correcting it. Remember that all test cases must be applied after a replacement to ensure program correctness.

In [dYG95] a cognitive framework of software debugging has been introduced. From the point of view of this paper finding and correcting bugs using model-based diagnosis can be viewed as isolation approach debugging. The comprehension approach to software debugging is done by

$$
\begin{tikzcd}
\Pi \arrow[r] & SD,COMP \arrow[dr] & & \\
& & \Delta \arrow[r] & St_1,\ldots,St_n \in \Pi \\
Tests \arrow[r] & OBS \arrow[ur] & &
\end{tikzcd}
$$

Figure 1.1: Mapping from Programs to Diagnosis Systems

using the programs semantics, the program itself and a specification. This approach does not evaluate the program to detect wrong behavior. The comprehension approach is also known as *code review*.

To combine the model-based diagnosis approach with software debugging we have to map programs to diagnosis systems. A diagnosis system consists of a set of components and a behavioral description. Adding observation to such a system allows the derivation of diagnoses (see 1.2.1 for an introduction). Conversion can be done by using program parts such as statements as diagnosis components and the semantics of the parts given by the programming language as behavioral description. Observations on the other hand are related to information about signal and variable values given for a specific test case. After computing diagnoses we must map diagnosis components back to the program parts. Figure 1.1 shows this process.

An automatic debugger must consists of a part converting programs and test cases to diagnosis systems and observations, a part (using standard model-based techniques) for doing diagnosis, and a part associating computed diagnoses to program bugs.

Mapping back from diagnoses to related program parts (e.g., statements) can be done by adding additional information to the original program. This information must include the statement which can be buggy and should also include the probability of the bug and how to correct it. For the previous example the output of a debugging tool should return the following output:

```
architecture BEHAV_2 of MEM is
begin
    MEM_UNIT: process
        variable mem:  std_logic_vector(1 to 4)
    begin
        if clk = '1' and de = '1' then -- UNDEFINED BUG(0.003)
            if rw = '1' then -- UNDEFINED BUG(0.003)
                mem := encrypt(data);
            else
                data <= decrypt(mem); -- BUG(0.02) REPLACE
                                BY decrypt_new(mem)
            end if;
        end if;
```

```
        wait on clk;
    end process;
end BEHAV_2;
```

To give a brief idea about how model-based diagnosis can be used for program debugging consider the following example:

```
    if flag then
        s := 0;
    else
        s := 5;
    end if;
```

If `s` is assumed to be 5 after program part execution, flag is set to true and `s` is therefore 0, a misbehavior occurs and should be related to the causing part of the program. In this case diagnosis is simple. Replacing `flag` by `not flag` is a solution. Although replacing `5` by `0` is a solution. Assume that `s` should be 3, if flag is true. In this case `not flag` does not explain the faulty behavior but replacing `0` by `3` does. What we have used to restrict the number of possible diagnosis is the knowledge about the program semantics.

Aim of this work is to show how the conversion of programs to diagnosis systems can be done. The proposed solutions are compared related to their diagnosis capabilities, extentiability, and applicability.

1.2 Basic Definitions

1.2.1 Model-based Diagnosis

In this section we introduce basic definitions some of them taken from [Rei87, GSW89, dKMR90] but based on Unger's master thesis *Ein Modellbasiertes Diagnosesystem - Erweiterung und Implementierung des Ansatzes nach Reiter* [Ung91]. Slight modifications and corrections of definitions and algorithms have been applied to [Ung91]. For an introduction and overview to model-based diagnosis and their application to the diagnosis of digital circuits see also [dKW87, DSH$^+$82, Dav84, DH88, Ham88, Ham91].

The use of model-based diagnosis is limited to systems where the following proberties hold. First the system must be composeable, i.e., must consists out of components and their interconnections. Secondly the behavior of the components should be known, i.e., a model must be available. Note that all preconditions are fulfilled for program debugging.

Definition 1.2.1 (Diagnosis System (Reiter)) *A (diagnosis) system is a pair* $(SD, COMP)$ *where:*

1. *SD is the system description, describing the behavior of components and their interconnections.*

2. *COMP, the system components, is a finite set of constants.*

The behavior of components must be defined at least in the case where a component is assumed to behave correct. It is also possible to define a faulty behavior. To do this a function is introduced associating components to behavior modes.

Definition 1.2.2 (Component Modes) *$MODES$ is a finite set of constants including at least* undef *and* ok.

The set $MODES$ contains all behavior modes for components. At least the two distinct modes $undef$ and ok must be element of $MODES$. For generality we also assume the existence of a function *modes* associating a subset of $MODES$ to every component. This subset must include $undef$ and ok.

$modes : COMP \mapsto 2^{MODES} \cup \{ok, undef\}$

Usually only the predicates ab or ok are used to indicate wether a component works correct or not. The relation between the component modes and the ab and single ok predicate is as follows: $undef$ has the same underlying meaning as ab or $\neg ok$, whereas ok can be expressed by $\neg ab$.

It is worth noting that the system description can describe the component behavior for the given modes, if one exists. It is not necessary that every mode has an associated behavior.

A diagnosis problem is stated by giving a system description and a set of observations. If the observations contradict the expected behavior there must be a subset of $COMP$ whose incorrectness would explain the observations.

Definition 1.2.3 (Diagnosis Problem) *A diagnosis problem is a triple $(SD, COMP, OBS)$ where SD is the system description of the system to be diagnosed, $COMP$ is the set of components in the system, and OBS is a finite set of observations.*

The goal of the diagnosis problem, formally defined below, is to determine the component mode for each component such that the observations are explained.

Definition 1.2.4 (Mode Assignment) *A mode assignment set MA is a set of predicates of the form $m(c)$ where $m \in MODES$ and $c \in COMP$.*

$MA \subseteq \{m(c) | m \in MODES \wedge c \in COMP\}$

A mode assignments associates a mode with a component. Intuitively, it should be clear that a mode assignment is related to a diagnosis.

Definition 1.2.5 *Let MA be a set of mode assignments. The function γ returns the set of components used in MA.*

$\gamma(MA) = \{c | \exists m \in MODES : m(c) \in MA\}$

For obvious reasons, every component should be assigned only to one mode, iff a diagnosis should be expressed. We therefore define:

Definition 1.2.6 (Exact Mode Assignment) *Let MA be a set of mode assignments. We call MA* exact *if no component is assigned two different modes.*

MA *is exact* $\Leftrightarrow |MA| = |\gamma(MA)|$

Now we can define what we mean when we speak about diagnosis.

Definition 1.2.7 (Diagnosis) *Let $(SD, COMP, OBS)$ be a diagnosis problem. A mode assignment Δ is a diagnosis ($DIAG(\Delta)$) iff $SD \cup OBS \cup \Delta \cup \{ok(c)|c \in COMP \setminus \gamma(\Delta)\}$ is consistent.*

Note that in the case where only two behavioural modes are available we also write $\{C_1, \ldots, C_n\}$ for the diagnosis $\{undef(C_1), \ldots, undef(C_n)\}$.

Task of a diagnosis engine is to determine the mode assignments for every component. As a matter of fact a simple diagnosis algorithm will need $|MODES|^{|COMP|}$ assumptions about mode assignments to find all diagnosis. Finding all diagnosis is therefore intractable. To avoid this problem, [Rei87] has introduced an algorithm allowing to order diagnosis search by the cardinality of the diagnoses. Only minimal diagnosis are searched for.

Definition 1.2.8 (Mode Assignment Subsumption) *Let MA_1, MA_2 be two exact mode assignments. We say that MA_1 subsumes MA_2 ($SUB(MA_1, MA_2)$) if the following hold:*

$\forall m(c) \in MA_1 : m(c) \in MA_2 \vee undef(c) \in MA_2$

The above definition prefers mode assignments with smaller cardinality and mode assignments not using the undefined behavior mode, i.e., it prefers modes which "explain more"[1].

Now we can define what minimality in this sense means.

Definition 1.2.9 (Minimal Diagnosis) *Let Δ be a diagnosis for $(SD, COMP, OBS)$. We say Δ is minimal iff there exists no diagnosis subsuming Δ.*

$MIN(\Delta) \Leftrightarrow \nexists \Delta' : DIAG(\Delta') \wedge SUB(\Delta', \Delta)$.

The aim is now to determine minimal diagnosis out of a given system and observations. To do this we can use a slightly modified version of Reiters [Rei87] diagnosis algorithm (corrected by [GSW89]).

Before describing the extended diagnosis algorithm we introduce the following definition used later:

[1] $undef$ explains not why a component may fails.

Definition 1.2.10 *Let MA be a set of mode assignments. The function ab answers a set of mode assignments which are element of MA indicating that the component behaves incorrectly.*

$ab(MA) = \{m(C) | m(C) \in MA, m \neq ok\}$

An interesting property of diagnosis using fault modes is that not all supersets of a diagnosis needs to be diagnoses, too. This is different to the case where only ab, $\neg ab$ respective $\neg ok$, ok modes are available (see [Rei87, dKMR90]) and can be simple explained by a two inverter chain.

Example 2:

Assume the inverters I_1, I_2 with associated modes ok for correct behavior, $undef$ for undefined (wrong) behavior, $s0$ for stuck at zero, and $s1$ for stuck at one. The system description can be formalized using first order logic:

$inverter(I) \rightarrow (ok(I) \rightarrow out(I) = not(in(I)))$

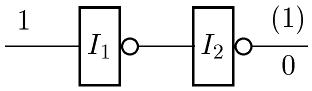
$inverter(I) \rightarrow (s0(I) \rightarrow out(I) = 0)$

$inverter(I) \rightarrow (s1(I) \rightarrow out(I) = 1)$

$inverter(I_1)$

$inverter(I_2)$

$out(I_1) = in(I_2)$



The expected value (1) is not measured at the output. Instead $false$ indicates a faulty behavior. Using the definition of minimality and the given fault modes, we can derive two minimal diagnoses $\{\{s1(I_1)\}\}$ and $\{\{s0(I_2)\}\}$. While $\{\{s1(I_1), s0(I_2)\}\}$ is a diagnosis, $\{\{s1(I_1), s1(I_2)\}\}$ is not.

□

Similar to [Rei87] we introduce conflicts and additionally choices. Choices can be viewed as disjunctive connection of mode assignments whereas conflicts represents conjunctive connections.

Definition 1.2.11 (Conflicts) *Let $(SD, COMP, OBS)$ be a diagnosis system and CO an exact mode assignments. CO is a conflict of $(SD, COMP, OBS)$ iff*

$SD \cup OBS \cup \bigcup_{m \in CO} \{m\}$

is inconsistent.

Additionally we define minimal conflicts as conflicts where no subset is a conflict. A conflict can be interpreted as follows. The mode assignments inside the conflicts represents assumptions about the behavior of the associated components which cannot be assumed at the same time because they imply a contradiction.

Choices are related to conflicts.

Definition 1.2.12 (Choice) *Let* $CO = \{a_1(c_1), \ldots, a_n(a_n)\}$ *be a conflict of a diagnosis system* $(SD, COMP, OBS)$*. A choice is a set* CH *of mode assignments where every* a_i *is replaced by the other modes available for* c_i*.*

$CH(CO) = \bigcup_{i \in \{1,\ldots,n\}} \{h_i(c_i) | h_i \neq a_i, h_i \in modes(c_i)\}$

Choices represents possible changes of a conflicts with the aim of finding an exact mode assignment which is not contradictory, i.e., is a diagnosis.

Reiter's diagnosis algorithm uses a hitting set tree for computing diagnosis. Greiner et al showed that improvements within the algorithm must be done to ensure correct diagnosis results. Instead of a hitting set tree a hitting set dag is used.

Definition 1.2.13 (Hitting Set) *A hitting set for a collection of choices* C *is a set* $H \subseteq \bigcup_{S \in C} S$ *such that* $H \cap S \neq \{\}$ *for each* $S \in C$*.*

A hitting set is said to be minimal if no set subsuming the hitting set is itself a hitting set.

The following theorem introduced an proven by Reiter gives the connection between hitting sets and diagnosis.

Theorem 1.2.1 (Reiter, 1987; Theorem 4.4) *A set* $\Delta \subseteq COMP$ *is a diagnosis for* $(SD, COMP, OBS)$ *iff* Δ *is a minimal hitting set for a collection of conflict sets and their related choices for* $(SD, COMP, OBS)$*.*

This theorem represents the extended version fitting in the introduced diagnosis framework.

The diagnosis algorithm presented in the remainder of this section assumes the existence of an ordered set of choices F, and an ordering S of nodes to be expanded. Since edges of the hitting set dag may have an *ok* label and we have the aim to expand nodes related to the number of *under* mode assignments, we expand nodes with the lowes number of *udef* mode assignments first. On the other hand we only want to compute minimal diagnoses, e.g., preferring mode assignments differing from the *undef* mode, and we want to expand nodes with an *undef* label of the previous edge at last.

The ordering S has the following appearance:

Number of *undef* mode assignments	Nodes
1	$n_{1,1}, n_{1,2}, \ldots$
2	$n_{2,1}, n_{2,2}, \ldots$
$\vdots$	$\ldots$

where $n_{i,j}$ is preferred over $n_{i,k}$, i.e., $j < k$, iff the number of $undef$ mode assignments for $H(n_{i,j})$ is less than for $H(n_{i,k})$. Note that other orderings can be defined, e.g., an ordering related to component fault probabilities.

Algorithm HS_DAG

1. Let D present the growing dag. Generate a node which will be the root of the dag. Add the node to the ordering S. This node will be processed in Step 2 below.
2. Process the ordering S by the number of $undef$ modes included in nodes label. To process a node n:
   (a) Define $H(n)$ to be the set of edge labels on the path in D from the root down to node n.
   (b) If for all $x \in F, x \cap H(n) \neq \{\}$ then label n by $\surd$. Otherwise label n by Σ where Σ is the first member of F for which $\Sigma \cap H(n) = \{\}$.
   (c) If n is labeled by a set Σ, then for each $\sigma(c) \in \Sigma$ where $c \notin \gamma(H(n))$, generate a new downward arc labeled by $\sigma(c)$. This arc leads to a new node m with $H(m) = H(n) \cup \{\sigma(c)\}$. The new node m will be processed after all nodes in the same generation (with the same number of ab mode assignments) as n have been processed.
3. Return the resulting dag D.

Improving HS_DAG

1. *Reusing Nodes*: This algorithm will not always generate a new node m as a descendant of node n. There are two cases to consider:
   (a) If there is a node n' in D such that $H(n') = H(n) \cup \{\sigma\}$, then let the σ-arc under n point to the the existing node n'.
   (b) Otherwise generate a new node as described in case 2c above.
2. *Closing*: If there is a node n' in D which is labeled by $\surd$ and $SUB(ab(H(n')), ab(H(n)))$, then close n. A label is not computed for n nor are any successor nodes generated. A closed node is labeled by $\times$.
3. *Pruning*: If a set Σ is to label a node n and it has not been used previously, then attempt to prune D as described in the following:
   (a) If there is a node n' which has been labeled by the set S' of F where $\Sigma \subset S'$, then relabel n' with Σ. For any $\alpha \in S' \setminus \Sigma$, the α-edge under n' is not longer allowed. The node connected by this edge and all of its descendants are removed, expect for those nodes whith another ancestor which is not being removed. This step may eliminate the node that is currently being processed.
   (b) Interchange the sets S' and Σ in the collection F which has the same effect as removing S' from F.

The **HS_DAG** algorithm can be used for diagnosis as described in [Rei87]. Instead of computing all conflict sets (and related choices) and storing it in the ordered set F, conflicts are only computed on demand. This can be expressed by changing the **HS_DAG** algorithm to:

2.(b)' If for all $x \in F, x \cap H(n) \neq \{\}$ then call the theorem prover $TP(SD, COMP, OBS, H(n))$. If the result is equal to $\surd$ then label n by $\surd$. If not, then label n by the result of the theorem prover call and add the result as last element to F. Otherwise label n by Σ where Σ is the first member of F for which $\Sigma \cap H(n) = \{\}$.

Although it is not necessary that the set F should contain only minimal choices, it is recommended to improve performance. The theorem prover call TP answers $\surd$ if the given assumptions are consistent and a (minimal) choice set otherwise.

$$TP(SD, COMP, OBS, H(n)) =$$
$$\begin{cases} \surd & \text{if} \begin{pmatrix} SD \cup OBS \cup \\ \{m(c) | m(c) \in H(n)\} \cup \\ \{ok(c) | c \in COMP \setminus \gamma(H(n))\} \end{pmatrix} \not\models \perp \\ CH(MIN(SD, COMP, OBS, H(n))) & \text{otherwise} \end{cases}$$

The function MIN computes minimal conflict sets.

Algorithm MIN(SD,COMP,OBS,A)

1. Create a new set A' containing all elements of A.
2. For every element x in A remove x from A'. If $SD \cup OBS \cup \{m(c) | m(c) \in A'\}$ is consistent add x to A'.
3. Return A' as result.

Computing minimal conflicts is fast since proving consistency only using a subset of $COMP$ is faster than using all elements of $COMP$.

Additionally we can include another closing rule to the **HS_DAG** algorithm to improve performance. In many cases the computed hitting set never leads to a diagnosis because it is inconsistent with the observations.

2.(b).(2) *Final Closing*: If the hitting set of node n is inconsistent with the system description and the observations, i.e., $SD \cup OBS \cup \{m(c) | m(c) \in H(n)\} \models \perp$, then close the node and do not compute a any successors. A final closed node is labeled by $\circ$.

The diagnosis algorithm can now be expressed by:

Algorithm DIAGNOSE(SD,COMP,OBS)

1. Construct a hitting set tree using algorithm **HS_DAG**.

2. All hitting sets leading to a vertex labeled by $\surd$ are diagnoses.

The time (and space) complexity of DIAGNOSE is given by the time (space) complexity of HS_DAG which is related to the number of created nodes.

Theorem 1.2.2 (Time complexity of DIAGNOSE) *Searching for all diagnoses using* **DIAGNOSE** *has a worst case time complexity of* $o((|MODES| \cdot |COMPS|)^{|COMPS|})$. *Searching for diagnoses of a size ds has a worst case time complexity of* $o((|MODES| \cdot |COMPS|)^{ds})$.

Note that this complexity result represents only an upper bound. In practice the algorithm performes much better.

> **Proof:** In the worst case the time complexity of **DIAGNOSE** is determined by the worst case time complexity of **HS_DAG**. By ignoring pruning rules and the costs for each theorem prover call the time complexity of **HS_DAG** is related to the number of created nodes. In the first step only one node is created. In the second step the number of created nodes is given by $|MODES| \cdot |COMPS|$, and so on. Searching for all diagnoses of a size less than ds has a time complexity of
>
> $$\sum_{i=0} ds(|MODES| \cdot |COMP|)^i = \frac{(|MODES| \cdot |COMP|)^{ds+1} - 1}{(|MODES| \cdot |COMP|) - 1}$$
>
> Since $(|MODES| \cdot |COMP|) >> 1$ the worst case time complexity is of order $o((|MODES| \cdot |COMP|)^{ds})$. ■

The theoretical time complexity of a diagnosis algorithm should be of order $|MODES|^{|COMP|}$. However **DIAGNOSE** has a much better average performance because diagnoses with a smaller size are much more likely. Finding all single diagnoses for example is proportional to $|MODES| \cdot |COMP|$.

The following example shows the extended algorithm in action.

Example 3: Assume the previous example using a chain of two inverters. After the first step, the root node n_0 is initialized $H(n) = \{\}$ and inserted into the ordering S.

$S_0 = \{0 \mapsto \{n_0\}\}$

The ordering is processed by choosing n_0, removing it from S and processing it. Since F is empty, a theorem prover call must be made answering the choice $\{s0(I_1), s1(I_1), undef(I_1), s0(I_2), s1(I_2), undef(I_2)\}$ because $\{ok(I_1), ok(I_2)\}$ is a conflict set. New nodes are created and inserted into the ordering S.

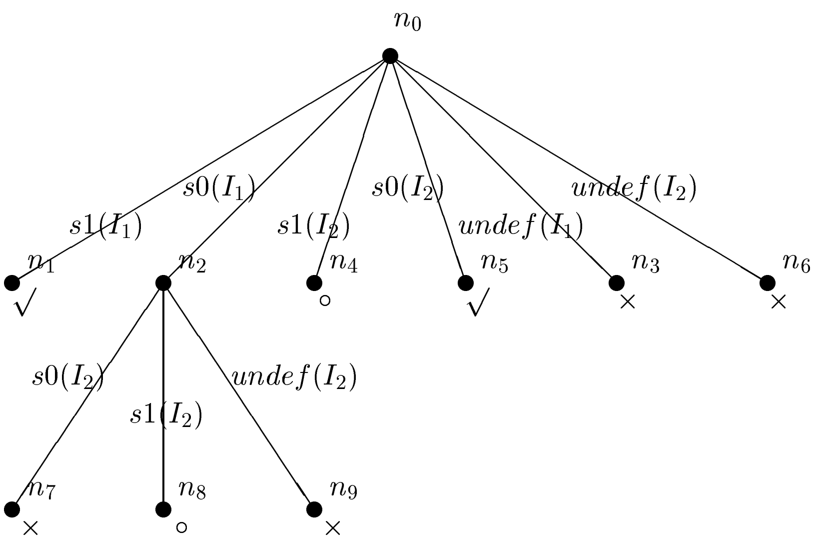
$S_1 = \{1 \mapsto \{n_1, n_2, n_4, n_5, n_3, n_6\}\}$

where

$H(n_1) = \{s1(I_1)\}$, $H(n_2) = \{s0(I_1)\}$, $H(n_3) = \{undef(I_1)\}$, $H(n_4) = \{s1(I_2)\}$, $H(n_5) = \{s0(I_2)\}$, $H(n_6) = \{undef(I_2)\}$.

The ordering S is processed and labels for the nodes $n_1, \ldots, n_6$ are computed. Because of the definition of subsumption and since $undef$ modes are used for computation after all other modes we close the nodes n_3 and n_6. The hitting set related to n_3 is subsumed by the hitting set of n_1.

The node n_4 can be finally closed since $SD \cup OBS \cup \{s1(I_2)\}$ is inconsistent. Only the node n_2 can be expanded. The label of n_2 is given by $\{s1(I_1), ok(I_1), undef(I_1), s0(I_2), s1(I_2), undef(I_2)\}$. Because I_1 has a mode assigned in a previous step only the nodes n_7, n_8 and n_9 with hitting sets $H(n_7) = \{s0(I_1), s0(I_2)\}$, $H(n_8) = \{s0(I_1), s1(I_2)\}$, $H(n_8) = \{s0(I_1), undef(I_2)\}$ are created. The same arguments as above lead to a final hitting set tree:



From this hitting set dag we can easily compute the diagnoses by going from the root node to the node labeled by $\surd$. The exact component mode sets $\{s1(I_1)\}$ and $\{s0(I_2)\}$ are minimal diagnoses.

□

We have implemented the above algorithm using Visualworks V2.0 (Parc Place Smalltalk). It has been shown that the algorithm **DIAGNOSE** is sufficiently for systems with several hundred components using 4 behavior modes. Finding all diagnosis needs only less than one minute on a Sparc 10 with 96 MB main memory.

Measurement Selection

Finally we introduce the concept of entropy-based measurement selection [dKW87]. The entropy H is given by probability values p_i:

$$H = -\sum p_i log p_i$$

In case of measurement selection we want to select those measurement point with the smallest entropy. Note that every diagnosis of a system $(SD, COMP, OBS)$ entails values for measurement points or says nothing about the value. So it is possible to classify all diagnoses regarding their value prediction.

Let S_{ik} be the set of all diagnoses predicting the value v_k for measurement point x_i, i.e., $SD \cup OBS \cup \{m(c)|m(c) \in \Delta_i\} \cup \{ok(c)|c \in COMP \setminus \gamma(\Delta_i)\} \models x_j = v_k$, and U_i be the set of diagnoses predicting no value for x_i.

To compute the entropy for x_i we must use the information given by all diagnoses leading to:

$$H_e(x_i) = \sum_{k=1}^{m} p(x_i = v_k) H(x_i = v_k)$$

where $v_k \in \{v_1, \ldots, v_m\}$ is the set of all possible values. To calculate this entropy we must know the value of $p(x_i = v_k)$. This value can be obtained by using the conditional probability $p(x_i = v_k|\Delta)$ where Δ is a diagnosis, and the probability of the diagnosis $p(\Delta)$.

$$p(x_i = v_k) = \sum_{\Delta} p(x_i = v_{ik}|\Delta) p(\Delta)$$

We know that $p(x_i = v_k|\Delta) = \begin{cases} 1 & \text{if } \ldots \cup \{m(c)|m(c) \in \Delta_i\} \cup \ldots \models x_j = v_k \\ 0 & \text{otherwise} \end{cases}$ and that the probability of a diagnosis candidate is given by $p(\Delta) = \prod_{m(c) \in \Delta} p_m(c) \prod_{c \in COMP \setminus \gamma(\Delta)} p_{ok}(c)$. The probabilities can vary for components and modes. $p_m(c)$ is assumed to store the fault probability value.

Since some candidates which are element of the set U_i also leads to a value v_k for x_i we cannot compute $p(x_i = v_k)$ exactly. So

$p(x_i = v_k) = \sum_{\Delta \in S_{ik}} p(\Delta) + \epsilon_{ik}$

where

$\sum_{k=1}^{m} \epsilon_{ik} = \sum_{\Delta \in U_i} p(\Delta)$.

Assuming that every value is equal likely we get:

$\epsilon_{ik} = \frac{\sum_{\Delta \in U_i} p(\Delta)}{m}$.

[dKW87] also showed how component probabilities can be changed using probabilities computed for measurement selection. Practical usage of the approach requires that we restrict the diagnosis set to the set of diagnoses of interest, e.g., the set of minimal diagnoses. De Kleer et al use all diagnoses which is not practical since the number of diagnoses is of order $|MODES|^{|COMP|}$ in the worst case.

1.2.2 Programming Languages

This section describes basic definitions related to programming languages which are used for diagnosis. First we define a programming language as tuple containing a set of logical sentences defining the syntax and a set of logical sentences defining the semantics.

Definition 1.2.14 (Programming Language) *A programming language $\mathcal{L}$ is a tuple (S, E) where S denotes a set of logical sentences regarding syntax and E contains logical sentences describing the behavior.*

Programs written in a programming language can be viewed as elements of the set of all possible programs given by the syntax of the language.

Definition 1.2.15 (Program) *Let $\mathcal{L} = (S, E)$ be a programming language. A program Π is an element of $\{x | S \models x\}$. The behavior of the program is given by E.*

It is assumed that the logical sentences describing the syntax define basic elements of the language and their connections. Basic elements are also called (basic) language constructs. In VHDL for example the language consists of constructs related to concurrent statements such as *process* or *concurrent signal assignment* statements, and of constructs related to sequential statements (*conditionals*, ...) and expressions.

The semantics define how the basic elements behave. We assume that every language has a variable object as basic element. The value of variable objects (variables) can be modified by other programming language elements. A program itself manipulates used variables regarding the semantics rules. We assume the existence of a function *elements* returning a set of programming language constructs used in the given program, e.g, statements, expressions, function calls, ..., and a function *vars* returning the variables used. It is clear that $vars(\Pi) \subseteq elements(\Pi)$ for a given program Π.

Consider example 1.1. The set of all program elements contains:

```
if clk = '1' and de = '1' then if rw = '1' then mem := data; else data <= mem;
end if; end if;
```

and

`clk = '1'` and so on. The set of all variables has `rw`, `clk`, ... as elements.

Since variable values are changed by programs we define the initial values of variables before program (element) execution as input and the final values after program (element) execution as output of a program (element). A test case for a program is given by inputs and expected outputs.

Definition 1.2.16 (Test case) *Let Π be a program written in $\mathcal{L}$. A test case T for Π is a tuple (I, O) where I the input associates values with every variable used in Π and O the expected output specifies those variable values which must be observed after program execution.*

We say that a program is correct wrt a given test case if all expected outputs are equal to the computed outputs. Otherwise we say that a program behave (or is) incorrect.

Programs and program elements can be compared with other programs or program elements, either by examining the syntax or by using the semantics of the language.

Definition 1.2.17 (Identical) *Each program element p is only identical to itself and not to every other element p_2 in the program even it is syntactical or semantical equivalent. Identity is expressed by $\equiv$.*

Definition 1.2.18 (Syntactically Equal) *Let p_1, p_2 be two program elements of the programs Π_1 respective Π_2, where Π_1 need not be different from Π_2. They are called to be syntactically equal iff they have the same syntax.*

$p_1 =_s p_2 \Leftrightarrow S(p_1) = S(p_2)$

Definition 1.2.19 (Semantically Equal) *Let p_1, p_2 be two program elements of the programs Π_1 respective Π_2, where Π_1 need not be different from Π_2. They are called to be semantically equal iff they produce the same output applying the same input.*

$p_1 =_e p_2 \Leftrightarrow \forall I : E(p_1, I) = E(p_2, I)$

Note that the three equality relations build a hierarchy on their relation classes $R_r = \{(x, y)|r(x, y) = true\}$

Corollary 1.2.3 $R_{\equiv} \subset R_{=_s} \subset R_{=_e}$

The program element identity and equalities can be naturally extended to be applicable for programs. We say that two programs are identical if they are the same, they are syntactically equal if they have the same number of parts and all parts are syntactically equal, and finally they are semantically equal if they produce the same output for all given inputs.

Consider example 1.1 once more. The statement `mem := data;` is identical to itself, and syntactically equal to `mem := data;`. It is further semantically equal to `mem := not(not(data));`.

Definition 1.2.20 (Replacement) *Let p and r be program elements with p used in Π. A replacement of p by r leads to a new program Π' where all elements except p are taken from Π and r is used instead of p.*

$\Pi[p|r]$

Note that the replacement must also take account of the interconnection of the program and the syntax of the used programming language. Replacements may change the behavior of programs.

Definition 1.2.21 *A replacement r of an element p used in Π is said to be evaluation neutral iff $\Pi =_e \Pi[p|r]$.*

Replacing a program element by an evaluation equal program part does not change the results:

Corollary 1.2.4 $\Pi =_e (\Pi|p \leftarrow r) \Leftarrow p =_e r$

The opposite is not true in general. Consider for example the program

```
y := x ;
y := 0 ;
```

Replacing `x` by `x + 1` does not change the output of the program.

The program introduced in example 1.1 can be corrected by using the replacement [`data <= decrypt(mem);`|`data <= decrypt_new(mem);`].

As noted previously in this section program, elements change variable values. We do not have specified how program elements are executed neither in which sequence they are executed. Both informations must be specified within the semantics description of the programming language. The sequence of program element execution can be described by an evaluation trace defined by:

Definition 1.2.22 (Evaluation Trace) *Let Π be a program written in $\mathcal{L}$, and $T = (I, O)$ a test case. An evaluation trace $\mathcal{E}$ is a sequence $< I, p_1, e_1, \ldots, p_n, O >$ where e_i denotes mappings of variables to values and p_i a set of all program elements used to compute e_i.*

Note that this definition can be used for concurrent languages (such as VHDL) as well as for sequential ones because of its generality.

The definition of evaluation traces can be used to define the notion of terminating programs.

Definition 1.2.23 (Program Termination) *Let Π be a program written in $\mathcal{L}$, and T a test case. We say that Π terminates on the test case T iff the associated evaluation trace is finite. If it is infinite the program does not terminate.*

Definition 1.2.24 (Termination Neutral) *A replacement r of a program part p used in Π is said to be termination neutral iff $\Pi[p|r]$ terminates for all inputs on which Π terminates .*

The replacement previously introduced for example 1.1 is termination neutral.

1.2.3 Program Verification versus Diagnosis

After having given some basic definitions related to programs and programming languages we now discuss problems of software engineering. The aim of software engineering is to build programs with as little resources as possible while making them reliable and easy to maintain. Testing, code review and verification is used to reach this aim. While testing uses multiple test cases to prove the correctness of the program, verification proves the correctness in a more general sense. Verifying a program means to prove that given specification rules are fulfilled. If this is not the case, the programmer usually is interested in finding the cause of the difference between the specification and the program. This is similar to the test case approach. However, current verification tools do not help in the actual bug finding process. The differences between verification and software debugging can be summarized as follows:

Verification Verification checks wether a given program meets its specifications or not. Since it is very unlikely (except for trivial programs) that the specifications is complete verified programs are not necessarily correct in general. However the probability of being correct increases while proving more general rules given by the specification.

Verification can be formalized as follows. Let $COND_I$ be conditions related to the programs input leading to $COND_O$, conditions on the output which must hold if the program is correct and the input conditions are fulfilled.

$COND_I \wedge \Pi \models COND_O$

Diagnosis After detecting discrepancies between the expected behavior given by test cases and the output computed by a program, the task of a program debugging algorithm is to determine those program parts causing the wrong behavior. In the second step someone is interested in finding replacements for buggy parts of the program, leading to programs with a correct behavior. In difference to programming, in software debugging we assume that the correct program is only a small variant of the original (wrong) program.

Software debugging can be formalized by:

$\exists T_i : E(\Pi, input(T_i)) \neq output(T_i) \Rightarrow$ search for $\Delta \subseteq parts(\Pi)$ explaining the difference.

or

$\exists T_i : E(\Pi, input(T_i)) \neq output(T_i) \Rightarrow$ search for replacements $r_1, \ldots, r_k$ resulting in $\forall T_i :$ $E((\Pi[p_1|r_1]\ldots[p_k|r_k]), input(T_i)) = output(T_i)$

Note that the replacements must be termination neutral.

1.3 An Overview of Automatic Program Debugging

The idea of using programs to find and correct faults in other programs is not new. It was initially presented by Shapiro in his dissertation *Algorithmic Program Debugging* ([Sha83]). He deals with bug identification, if a program behaves incorrectly, and bug fixing, once one has been identified. For this purposes Shapiro introduces several algorithms using programs consisting only out of procedure calls. Because of this limitation his approach seems to be only accurate for logic programs, e.g., programs written in Prolog. The proposed algorithms solve the problem of finding a bug under the following circumstances:

1. A program terminates with a wrong output.
2. A program does not terminate.

Because the failure of a program can be caused by a given procedure call or by a missing one Shapiro introduces two different algorithms handling each case for seperately. Termination is detected by reaching a maximum computation depth given by the user.

In the second part of Shapiro's dissertation, he showed how detected faults can be fixed, i.e., repairs computed. His proposal possess several drawbacks. First, Shapiro's algorithms need heavy user (oracle[2]) interaction. Secondly, although Shapiro argues that

Although the target programs of the diagnosis algorithms are logic programs, the methods they employ are general, and are applicable to a large class of programming languages.

this seems not to be the case. The restriction to procedure calls and not looking inside the procedures definition is to strong. The last drawback is that Shapiro's algorithms though written in Prolog are not declarative at all.

Based on Shapiro's, work Ferrand [Fer87] and later Lloyd [Llo87] presented algorithms for bug detection in purely logic programs. Both approaches focus only on finding bugs once an incorrect behavior of a terminating program has been detected. Similar to Shapiro they introduce two predicates (algorithms), one for handling missing clauses and the other for finding incorrect ones. However, their algorithms are purely declarative. The drawback of needing user interaction still remains in both papers.

To avoid heavy user interaction, [CFD93] tried to solve the debugging problem by introducing model-based ideas to this domain. [CFD93] argue that the overall user interaction of the model-based approach is less than with Shapiro's. The main idea in applying model-based techniques lies in presenting the concept of elimination and addition of clauses to ensure a correct program behavior. A heuristic has been proposed to restrict the search space. However, no implementation or practical results have been published.

Based on [CFD93], Bond [Bon94] described an algorithm and results on applying model-based diagnosis to the logic programming domain. In [BP94], Bond et.al. further critically reviewed [CFD93], detected some weaknesses, and proposed a solution to these problems. Another reason for applying model-based diagnosis to the software domain lies in proving that the approach is indeed a general one, and can be used for all diagnosis problems, which has been shown, too. But so far the basic concepts of model-based diagnosis have rarely been applied to solve practical problems. This holds especially in the software domain where little effort has been spent in investigating new algorithms and concepts for diagnosing programs written in "real" programming languages like C++, Smalltalk, Ada, and so on.

From 1994 to 1995 Siemens Austria has funded the *Design Diagnosis of VHDL Specifications* (DDV) project with the aim of assisting hardware designers finding bugs in very large VHDL programs. In [FSW95, FSW96, SW96c] the application of model-based techniques for debugging programs written in the hardware description language VHDL (Very High Speed Integrated Circuit Hardware Description Language) has been described. The papers introduce tools for supporting the hardware design process. The underlying ideas can be found in this dissertation.

However, the initially proposed language model used in the DDV project has several drawbacks, regarding the model structure, which leads to unsatisfactory results in some cases. To avoid this problems [SW96b, SW96a] introduces a more sophisticated model, causing more computations.

Based on work done during the DDV project, Pawlin ([Paw96]) has implemented a program allowing to diagnose a subset of C. This master's thesis shows that model-based diagnosis can

[2]Although the output of programs can be checked by comparing the output with a formal or informal specification, it can be also done by using a second program or as previously stated by asking the user. The term oracle subsumes this three possibilities.

be efficiently used for certain aspects of imperative languages. Limits and problems are discussed but many problems, e.g., diagnosis of loops, remain unsolved.

Other approaches of program debugging include the representation of programs as devices with an associated functional model [GC89, AC91, SSGP92, IC92, Liv94]. [GC89] show the application of functional models to solve problems regarding design and redesign. In [AC91] based on [Joh86, Mur88] functional models are introduced to the program debugging domain. Diagnosis is done by tracing back from faulty behavior, but is not limited to logic programs. Differentiation between faults must be done by asking the user about expected values. On the other hand the approach needs a functional model which must be created and cannot be derived from the program in general. [Liv94] applies the approach to sequential imperative programs where the source code is not available. However, all those approaches fail in minimizing user interaction and/or automatic derivation of models related to the original program which can be used for diagnosis.

In [CWH94] a fault detection and correction algorithm for gate-level implementations of digital systems is presented. Because of the restriction to gate-level programs it can only be used for a very small subset of VHDL. In addition only single faults can be corrected. This approach is very similar to the diagnosis of hardware and has only a small impact to the software domain.

Another approach to program debugging introduced by Burnell et al [BH93, BH95] uses probability measurements to guide diagnosis. The program debugging process is divided into parts. In the first one, program parts that may cause a discrepancy are computed. This is done by tracing the incorrect output back to the inputs and collecting the involved statements. To improve the diagnosis result, i.e., focus on the most probable statements, a belief network (Bayesian Network) is used. Statement types and fault types are related by a probability which is given by an "expert". This probability can be used to compute a probability of failure for every statement or execution path. Burnell et al, present the result that their program diagnosis is about 75 percent correct. This value is only related to the diagnosis of one program and it is not very likely that diagnosing another program will also lead to such a high value. This proposition can be argued by saying that a general fault probability which can be assigned to every component depending neither on the programmer nor on the specific programming language used, is very unlikely to find. Therefore the approach of Burnell et al can only be used for maintaining an existing (large) program. Using the same approach for another one requires a recomputation of probabilities and a fine tuning.

Chapter 2

Syntax and Semantics of VHDL

2. Was der Fall ist, die Tatsache, ist das Bestehen von Sachverhalten.

Ludwig Wittgenstein, 1921
Tractatus logico-philosophicus
Logisch-philosophische Abhandlung

In the following subsections we present the syntax and a denotational semantics for a subset of VHDL called **RVHDL** . This subset allows to present the main ideas but avoids too much concentration on VHDL-specific features. For a syntactical and semantical definition of full VHDL see [VHD88]. Other (restricted) VHDL subsets and their formal semantics can be found in [KB95] and [BGM94]. An introduction to VHDL and how to use it in circuit design can be found in [Nav93, Bha92].

2.1 Syntax of RVHDL

The syntax of **RVHDL** can be found in appendix A.

The used VHDL subset contains only the concurrent process statement. Apart from the assertion statement, all other concurrent statements can be converted into semantically equal process statements. We also avoid the usage of records, type definitions, type conversion, alias declarations, procedures and n-dimensional arrays. We only allow the usage of built-in types (`bit`, `boolean`, and `integer`), the types `std_logic` and `std_logic_vector` and their associated functions. Note that all propositions in this thesis, can be extended to full VHDL, but with additional modelling effort as the result. A more serious restriction of **RVHDL** is that we only allow one wait statement at the end of the statement part of a process statement. This restriction simplifies the semantic model introduced in section 2.2. However a large class of VHDL programs can be converted into **RVHDL** .

We assume that the underlying semantics of **RVHDL** is the same as for full VHDL. Note that not all syntactical restrictions imply restrictions in expressiveness compared to (full) VHDL. Restrictions can be summarized as follows.

1. Allowing only one `wait` statement at the end of a process restricts the expressability of **RVHDL** . However most tasks solvable using VHDL can also be written in **RVHDL** , except in the case where multiple `wait`'s are necessary. Programs for direction detection of `inout` signals use multiple `wait`'s.
2. **RVHDL** does not allow the usage of records, n-dimensional arrays, aliases, type conversion and user-defined procedures. The usage of one-dimensional arrays and user-defined functions is allowed.
3. Only one entity and architecture statement can be used. By recursive replacement of component instantiation statements by their architecture code and using the given port map, every (full) VHDL program can be transformed into a semantically equivalent **RVHDL** program if the other restrictions are taken into account.

Most of the above restrictions are also used for register transfer level (RTL) programs. RTL programs allow the automatic conversion to the gate level. Because of this fact, most of todays VHDL programs are RTL programs.

2.2 Semantics of RVHDL

In this section we define the a denotational semantics for **RVHDL** . The syntax of **RVHDL** and also VHDL gives the basic structure of a VHDL program. It consists of an entity with associated architecture that declares the signals used. We assume that a set SIG contains all signals declared in the entity port or architecture declarative part. Since architectures contain processes we also assume the existence of a set $PROCESS$ containing all processes. Processes on the other hand may change signal values. For every signal used as target in a signal assignment statement within a process, there is a special signal called its *driver*. Driving signals are assumed to be stored in the set $DRIVERS$ ($DRIVERS \cap SIG = \emptyset$). Every element of $DRIVERS$ can be denoted as S_P where S is the target signal of a signal assignment statement which is element of the statement block of process P. The value of the driver is given by executing the process, while the value of the related signal is determined by the driver and the given resolution function. Because VHDL permits that every signal can have multiple drivers, this fact must be taken into account. To make value computation deterministic, VHDL introduces the concept of resolution function. The value of the original signal (which is element of SIG) can be computed using the resolution function and all driving signals (drivers). For simplicity we therefore introduce a function $Drivers$ associating drivers with signals, and a function $Target$ associating drivers with processes.

$Drivers : SIG \mapsto 2^{DRIVERS}$

$Target : PROCESS \mapsto 2^{DRIVERS}$

Processes also use variables to store information. To reflect this we introduce the function $Vars : PROCESS \mapsto 2^{VARS}$ associating processes with the used variables.

Because user-defined functions can be used, we introduce a variable $FUNC$ storing the names of all functions (built-in and user-defined) and a function environment storing the associated function bodies only for user-defined functions. For simplicity we assume that every function has an unique name. The function environment $FEnv : FUNC \mapsto$ **RVHDL** is assumed to return the statements used in the body, i.e., $FEnv(\texttt{decrypt}) =< St_1, \ldots, St_n >$. In addition we assume the existence of a function $Args$ answering the (ordered) list of function arguments.

After describing the static structure of a VHDL program, we are interested in the dynamic behavior.

From the LRM we know that signals (used for communication between processes) have associated projected output waveforms (POW), which store the future values. This POW is updated by processes whenever a VHDL statement tries to change a signal value. Note that the signal value is not changed during the evaluation of sequential statements in a process. Only variable values can be changed directly by variable assignment statements. Before defining exactly how POW's can be updated, we introduce the second necessary data structure, *environments*. Environments store values of variables and the actual values of signals.

Definition 2.2.1 (Environments) *Assume that there exists a function domain (domain :* $VAR \cup SIG \cup DRIVERS \mapsto 2^{VALUE}$*) associating variables and signals with value domains. An environment is a function associating variables or signals with values.*

$Env : VAR \cup SIG \cup DRIVER \mapsto VALUE$

The set of all environments is denoted by ENV. We assume that if a value of a variable or signal is not known the environment function answers the undefined value ϵ which is no element of $VALUE$. We define an operation remove ($-$), add ([ ]) to allow manipulation of environments as follows:

$Env - O = Env'$ with $Env'(O') = Env(O')$ if $O \neq O'$ and $Env'(O) = \epsilon$ otherwise.

$Env[(O, V)] = Env'$ with $Env'(O') = Env(O')$ if $O \neq O'$ and $Env'(O) = V$

POW's store values over time. We therefore define a POW as a set of tuples containing values and time points. The semantics of a tuple for a POW of signal S is to change the actual value of signal S at the given time to the given value.

Definition 2.2.2 (Projected Output Waveform (POW)) *A projected output waveform (POW) Wav for a signal S is a set of tuples (V, T) where V denotes a value and T a time point. Every time point has one tuple in Wav at the maximum.*

$Wav \subseteq \{(V, T) | V \in domain(S), T \in Time, \nexists (V', T) \in Wav : V' \neq V\}$

The set of all POW's is denoted by WAV. We now define operations on POW's which are used for describing the semantics of VHDL. Accessing a POW Wav is done using the access function $Wav : TIME \mapsto VALUE$ defined by $Wav(T) = V \Leftrightarrow (V, T) \in Wav$. Adding new values at time points to a POW is referred to as adding a transaction. Because VHDL knows two forms of changing a signal value at a future time, we must differentiate between them. The first one is used if the keyword `transport` is used in a signal assignment statement while the second one called *inertial* is implied if this keyword is missing.

Adding a new transaction to a POW with *inertial* delay is done using the operation $[\ ]_{INERT}$. Adding it with *transport* delay is done using the operation $[\ ]_{TRANS}$. They are defined as follows.

$Wav[(V,T)]_{INERT} = \{(V',T')|(V',T') \in Wav, T' < T, V' = V\} \cup \{(V,T)\}$

$Wav[(V,T)]_{TRANS} = \{(V',T')|(V',T') \in Wav, T' < T\} \cup \{(V,T)\}$

While VHDL programs are executed, time increases and POW's must be updated to reflect time changes. Let us assume that $ActualTime$ always answers the actual time during program execution, that there exists a function $SEnv$ returning the actual value at the actual time for a given signal, and that we have also a function Pow answering the actual POW for the given signal. Then we define the integrity constraint on POW's which must be hold for all times $ActualTime$ as follows:

$\forall S \in SIG : \forall (V,T) \in Pow(S) : T \geq ActualTime.$

Therefore we must be able to remove tuples from every POW. An operation answering the minimum time for a given POW will be also useful.

$Wav - T = \{(V',T')|(V',T') \in Wav, T' > T\}$

$time(Wav) = min\{T|(V,T) \in Wav\}$

We describe the semantics of VHDL using a function that changes projected output waveforms and environments.

Let $PActive : PROCESS \mapsto \{true, false\}$ be a function indicating whether a process is active or not, $SActive, SEvent : SIG \cup DRIVERS \mapsto \{true, false\}$ be a function indicating whether a signal is active or whether an event occured. The function $PENV : PROCESS \mapsto ENV$ associates processes with their actual (variable) environment, and $SEnv :\mapsto ENV$ returns an environment for storing actual signal values.

Since signals can be influenced by several processes, we have introduced (like the VHDL Language Reference Manual LRM (see [VHD88])) a function $Drivers$ associating driving signals (related to processes) with signals defined in an entity or architecture. The value of a signal is determined by the values of the driving signals and the given resolution function. This resolution function is determined by the type of the signal. We assume a function *Resolution* associating the resolution function with a signal. We assume further that every signal defined in an entity or architecture has an associated resolution function. If it is not given by the type, we assume that the identity function is used.

$Resolution : SIG \mapsto FUNC$

$\forall S \in SIG : |Drivers(S)| = 1 \Rightarrow Resolution(S) = I$ where $I(X) = X$

$\forall S \in SIG : |Drivers(S)| > 1 \Rightarrow Resolution(S) = Resolution(type(S))$

The value of S is given by

$Env(S) = Resolution(S)(Env(S_1), \ldots, Env(S_n))$ where $S_i \in Drivers(S)$

Note that values of signals (element of SIG or $DRIVERS$) are given by the environment at the actual time during program execution.

The semantics of VHDL is given by the LRM [VHD88]. The execution of a VHDL program is divided into two phases, the initialization phase and the simulation cycle (LRM section 12.6.3). During the initialization phase values are associated with signals, $SActive, SEvent$ is set to true, the actual time is initialized to be 0ns, and for each process $PActive$ is also set to true. Finally, all processes are executed until they suspend.

$exec(Prog) = exec(Prog, ActualTime, PActive, SActive', SEvent', Pow', SEnv)$

where

$ActualTime = 0ns$

$$\begin{aligned}
&\forall P \in PROCESS : PActive(P) = false \wedge \\
&\qquad \forall V \in Vars(P) : PEnv(P)[(V, initialValue(V))]
\end{aligned}$$

$$\begin{aligned}
&\forall S \in SIG : \\
&\qquad SActive(S) = true \wedge \\
&\qquad SActive'(S) = false \wedge \\
&\qquad SEvent(S) = true \wedge \\
&\qquad SEvent'(S) = false \wedge \\
&\qquad SEnv(S) = initialValue(S)
\end{aligned}$$

$$\forall D \in DRIVERS : Pow(D) = \{\}$$

$$\begin{aligned}
Pow' = {} & execActiveProcesses(PROCESS, \\
& \qquad ActualTime, PActive, SActive, SEvent, Pow, SEnv)
\end{aligned}$$

The initial value of a signal is implicitly given by the first value of the domain of the signal, or explicitly by an expression located in the signal declaration.

After the initial phase a simulation cycle starts. If no driver is active, the simulation cycle advances to the next time a driver becomes active or a process resumes. After this each active signal is updated. Each process which is sensitive to a signal where an event occurs is set to active. Each active process is executed until it suspends. Program execution stops when time reaches a prespecified maximum time. In this case no further simulation cycles are started. Note that processes can become active because of an event on a signal, or because a condition is fulfilled.

$$\begin{aligned}
&exec(Prog, ActualTime, PActive, SActive, SEvent, Pow, SEnv) = true \\
&\qquad \text{if } ActualTime' \geq maxTime
\end{aligned}$$

$$\begin{aligned}
&exec(Prog, ActualTime, PActive, SActive, SEvent, Pow, SEnv) = \\
&\quad exec(Prog, ActualTime', PActive', SActive', SEvent', Pow', SEnv') \\
&\qquad \text{if } ActualTime' < maxTime
\end{aligned}$$

The new values for each simulation cycle, e.g., $PActive', ActualTime', \ldots$ are computed as follows:

1. *Should time advance?*

$$
\begin{aligned}
&\forall D \in DRIVER: \\
&\qquad \Big((time(Pow(D)) = ActualTime \rightarrow \\
&\qquad\qquad SActive'(D) = true \wedge \\
&\qquad\qquad SEnv' = SEnv[(D, Pow(D)(ActualTime))] \wedge \\
&\qquad\qquad Pow(D)' = Pow(D) - ActualTime) \vee \\
&\qquad (time(Pow(D)) > ActualTime \rightarrow \\
&\qquad\qquad SActive'(D) = false)\Big)
\end{aligned}
$$

$$\forall D \in DRIVER: SActive'(D) = true \wedge SEnv(D) \neq SEnv'(D) \rightarrow SEvent'(D) = true$$

$$\forall D \in DRIVER: SActive'(D) = false \rightarrow SEvent'(D) = false$$

$$
\begin{aligned}
&\Big((\forall D \in DRIVER: SActive'(D) = false) \wedge \\
&(\forall P \in PROCESS: PTime(P) > ActualTime)\Big) \rightarrow \\
&\qquad ActualTime' = min(\{T | T = time(Pow(D)), D \in Drivers(S), S \in SIG\} \cup \\
&\qquad\qquad\qquad \{T | T \in PTime(P), P \in PROCESS\}) \\
&\Big((\exists D \in DRIVER: SActive'(D) = true) \vee \\
&(\exists P \in PROCESS: PTime(P) = ActualTime)\Big) \rightarrow \\
&\qquad ActualTime' = ActualTime
\end{aligned}
$$

$$\forall P \in PROCESS: PTime(P) = ActualTime \rightarrow PTime'(P) = 0$$

$$\forall P \in PROCESS: PTime(P) > ActualTime \rightarrow PTime'(P) = PTime(P)$$

2. *Update Signal values*

$$
\begin{aligned}
&\forall S \in SIG: \\
&\qquad \Big((\nexists D \in Drivers(S): \\
&\qquad\qquad SActive'(D) = true \rightarrow SActive'(S) = false) \vee \\
&\qquad (\exists D \in Drivers(S): \\
&\qquad\qquad SActive'(D) = true \rightarrow SActive'(S) = true)\Big)
\end{aligned}
$$

$$
\begin{aligned}
&\forall S \in SIG: \\
&\qquad \Big((SActive'(S) = false \rightarrow SEnv'(S) = SEnv(S)) \vee \\
&\qquad (SActive'(S) = true \rightarrow \\
&\qquad\qquad SEnv'(S) = Resolution(S)(SEnv(D_1), \ldots, SEnv(D_n)))\Big) \\
&\qquad\qquad\qquad \text{where } D_i \in Drivers(S)
\end{aligned}
$$

$$
\begin{aligned}
&\forall S \in SIG : \\
&\quad \big((SEnv(S) = SEnv'(S) \rightarrow SEvent'(S) = false) \vee \\
&\quad (SEnv(S) \neq SEnv'(S) \rightarrow SEvent'(S) = true)\big)
\end{aligned}
$$

3. *Update processes*

   Processes are said to be active iff an event occurs on a signal which is in the process's sensitivity list, or the process condition evaluates to true ($Cond : PROCESS \mapsto EXPR$), or time advances to the process wait time ($PTime : PROCESS \mapsto TIME$). The process wait time t is given by the wait statement `wait for` t.

$$
\begin{aligned}
&\forall P \in PROCESS : \\
&\quad \Big(\Big(\Big((\forall S \in Sensitive(P) : SEvent'(S) = false) \wedge \\
&\quad (eval_E(Cond(P), SEnv') = false) \wedge \\
&\quad (PTime(P) \neq ActualTime')\Big) \\
&\qquad \rightarrow PActive'(P) = false\Big) \vee \\
&\quad \Big(\Big((\exists S \in Sensitive(P) : SEvent'(S) = true) \vee \\
&\quad (eval_E(Cond(P), SEnv') = true) \vee \\
&\quad (PTime(P) = ActualTime')\Big) \\
&\qquad \rightarrow PActive'(P) = true\Big)\Big)
\end{aligned}
$$

4. *Execute all active Processes*

$$
\begin{aligned}
&Pow' = execActiveProcesses(PS, ActualTime', SActive', SEvent', SEnv', Pow) \\
&\qquad \text{where } PS = \{P | P \in PROCESS, PActive'(P) = true\}
\end{aligned}
$$

All active processes are evaluated by successive execution of each individual process. Only the drivers POW's are updated during this part of the simulation cycle, signal values are not.

$$
\begin{aligned}
&execActiveProcesses([P_1, \ldots, P_n], ActualTime, SActive, SEvent, SEnv, Pow) = \\
&\quad exec([P_2, \ldots, P_n], ActualTime, SActive, SEvent, SEnv, Pow') \\
&\qquad \text{where } Pow' = eval(P, SEnv \cup PENV(P), Pow)
\end{aligned}
$$

$$
execActiveProcesses([], ActualTime, SActive, SEvent, SEnv, Pow) = Pow
$$

2.2.1 Process Evaluation

Sequential statements (*Conditionals*, *Signal Assignments*, *Variable Assignments*, ...) used in processes determine the process behavior.

Let $P = Id$: `process ... begin` $St_1 \ldots St_n$ `end process` be a process where St_i denotes a sequential statement.

The process evaluation is defined as follows:

$eval(P, Env, Pow) = eval(statements(P), Env, Pow)$

- *Sequential Statement Blocks*

  $$eval([St_1, \ldots, St_n], Env, Pow) = \\ \quad eval(St_2, \ldots, St_n, Env', Pow')$$

  where $(Env', Pow') = eval(St_1, Env, Pow)$

- *Signal Assignments*

  $$eval(T \texttt{ <= } E \texttt{ after } TE, Env, Pow) = \\ \quad (Env, (Pow(T))[(eval_E(E, Env), ActualTime + eval_E(TE, Env)]_{INERT})$$

  $$eval(T \texttt{<= transport } E \texttt{ after } TE, Env, Pow) = \\ \quad (Env, (Pow(T))[(eval_E(E, Env), ActualTime + eval_E(TE, Env)]_{TRANS})$$

- *Variable Assignments*

  $eval(T := E, Env, Pow) = (Env[(T, eval_E(E, Env))], Pow)$

- *Conditionals*

  $$eval(\texttt{if } E \texttt{ then } SB_1 \texttt{ else } SB_2 \texttt{ end if}, Env, Pow) = \\ \quad eval(SB_1, Env, Pow) \text{ iff } eval_E(E, Env) = true$$

  $$eval(\texttt{if } E \texttt{ then } SB_1 \texttt{ else } SB_2 \texttt{ end if}, Env, Pow) = \\ eval(SB_2, Env, Pow) \text{ iff } eval_E(E, Env) = false$$

- *Loops*

  $$eval(\texttt{while } E \texttt{ loop } SB \texttt{ end loop}, Env, Pow) = \\ \quad eval(\texttt{while } E \texttt{ loop } SB \texttt{ end loop}, Env', Pow') \text{ iff } eval_E(E, Env) = true \\ \qquad \text{where } (Env', Pow') = eval(SB, Env, Pow)$$

  $$eval(\texttt{while } E \texttt{ loop } SB \texttt{ end loop}, Env, Pow) = \\ \quad (Env, Pow) \text{ iff } eval_E(E, Env) = false$$

- *Null Statements*

  $eval(\texttt{null}, Env, Pow) = (Env, Pow)$

- *Wait Statement*

  $eval(\texttt{wait } \ldots, Env, Pow) = (Env, Pow)$

2.2.2 Expression Evaluation

After defining how VHDL programs and processes are executed, we introduce how expressions are evaluated. Expression evaluation needs the definition of types, their domains and defined operations. For the purposes of this thesis we assume only the existence of the types `boolean`, `integer`, `bit`, `std_logic`, and `std_logic_vector`. More about the functionality and domains can be found in the section 2.2.3.

We describe the semantics of the expression using the function $eval_E$ defined on expressions, relations, terms, factors and primaries as follows:

$eval_E(E\ \texttt{op}\ E, Env) = op(eval_E(R_1, Env), eval_E(R_2, Env))$

where $op \in \{and, or, xor, nand, nor, =, /=, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, +, -, *, /, \&, mod, rem, **\}$

$eval_E(\texttt{si}\ E, Env) = si(eval_E(E, Env))$

where $si = \{+, -\}$

$eval_E(\texttt{abs}\ P, Env) = abs(eval_E(P, Env))$

$eval_E(\texttt{not}\ P, Env) = not(eval_E(P, Env))$

$eval_E(\texttt{(}\ E\ \texttt{)}, Env) = eval_E(E, Env)$

If I denotes a function call, then the evaluation is defined by:

$eval_E(I\texttt{(}\ E_1, \ldots, E_n\ \texttt{)}, Env) = eval_E(E, Env'')$ where $Env'(A_i) = eval_E(E_i, Env)$ for all $A_i \in Args(I) =< A_1, \ldots, A_n >$, $[St_1, \ldots, St_n, \texttt{return}\ E] = FEnv(I)$, and $(Env'', Pow) = eval([St_1, \ldots, St_n], Env', \{\})$.

If I denotes an array access, then evaluation is done by:

$eval_E(I\texttt{(}\ E\ \texttt{)}, Env) = Env(I)[eval_E(E, Env)]$

The evaluation of variables, constants and signals is defined by:

$eval_E(I, Env) = I$ iff $I \in CONST$

$eval_E(V, Env) = Env(V)$ iff $V \in VARS \cup SIG$

2.2.3 Types and their Operations

As stated in the previous subsection, we only consider the types `boolean`, `integer`, `bit`, `std_logic`, and `std_logic_vector`. Whereas `boolean`, `integer`, and `bit` are VHDL predefined types, `std_logic` and `std_logic_vector` are not. For this work it is not relevant to make a difference. All types are described by their domain and their operations.

Because of the previous definition that environments need not to be totally defined, we assume that if an argument has the (evaluated) value ϵ, the result of the function is ϵ. This assumption holds for all built-in operations.

VHDL Type `boolean` The `boolean` type is given by the domain

$domain(\texttt{bit}) = \{true, false\}$

and the operations

$operation(\texttt{boolean}) = \{and, or, not, xor, nand, nor\}$.

The semantics of the operations are as expected.

and	false	true
false	false	false
true	false	true

or	false	true
false	false	true
true	true	true

xor	false	true
false	false	true
true	true	false

nand	false	true
false	true	true
true	true	false

nor	false	true
false	true	false
true	false	false

not	
false	true
true	false

No resolution function is defined for signals of type `boolean`.

VHDL Type `integer` The domain of the `integer` type is defined to be $[-32768, 32767]$. The operations $\{+, -, *, /, mod, rem, abs\}$ defined on `integer` behave like expected. No resolution function is defined for `integer`.

VHDL Type `bit` The domain of the `bit` type is given by

$domain(\texttt{bit}) = \{'0', '1'\}$.

$'0'$ can be viewed as false and $'1'$ as true. Operations are

$operation(\texttt{bit}) = \{and, or, not, xor, nand, nor\}$.

The semantics of the operations are as expected.

and	'0'	'1'
'0'	'0'	'0'
'1'	'0'	'1'

or	'0'	'1'
'0'	'0'	'1'
'1'	'1'	'1'

xor	'0'	'1'
'0'	'0'	'1'
'1'	'1'	'0'

nand	'0'	'1'
'0'	'1'	'1'
'1'	'1'	'0'

nor	'0'	'1'
'0'	'1'	'0'
'1'	'0'	'0'

not	
'0'	'1'
'1'	'0'

No resolution function is defined for signals of type `bit`.

VHDL Type `std_logic` The type `std_logic` uses 9 constants

$domain(\text{std\_logic}) = \{'U', 'X', '0', '1', 'W', 'L', 'H', 'Z', '-'\}$

with the following underlying meaning:

std_logic value	Description
'U'	Uninitialized
'X'	Strong unknown
'0'	Strong low
'1'	Strong high
'W'	Weak unknown
'L'	Weak low
'H'	Weak high
'Z'	Three-state
'-'	Don't care

Operations are

$operation(\texttt{std\_logic}) = \{and, or, not, xor, nand, nor\}$.

The semantics of the operations is defined by:

and	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'0'	'U'	'U'	'U'	'0'	'U'	'U'
'X'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'Z'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'W'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'L'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'H'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'-'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'

or	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'1'	'U'	'U'	'U'	'1'	'U'
'X'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'0'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'
'Z'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'W'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'L'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'H'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'
'-'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'

xor	’U’	’X’	’0’	’1’	’Z’	’W’	’L’	’H’	’-’
’U’	’U’	’U’	’U’	’U’	’U’	’U’	’U’	’U’	’U’
’X’	’U’	’X’	’X’	’X’	’X’	’X’	’X’	’X’	’X’
’0’	’U’	’X’	’0’	’1’	’X’	’X’	’0’	’1’	’X’
’1’	’U’	’X’	’1’	’0’	’X’	’X’	’1’	’0’	’X’
’Z’	’U’	’X’	’X’	’X’	’X’	’X’	’X’	’X’	’X’
’W’	’U’	’X’	’X’	’X’	’X’	’X’	’X’	’X’	’X’
’L’	’U’	’X’	’0’	’1’	’X’	’X’	’0’	’1’	’X’
’H’	’U’	’X’	’1’	’0’	’X’	’X’	’1’	’0’	’X’
’-’	’U’	’X’	’X’	’X’	’X’	’X’	’X’	’X’	’X’

not	
’U’	’U’
’X’	’X’
’0’	’1’
’1’	’0’
’Z’	’X’
’W’	’X’
’L’	’1’
’H’	’0’
’-’	’X’

The definition for `nand` and `nor` can be computed using the truth tables for `and`, `or`, and `not` and are therefore omitted.

`std_logic` has a resolution function $Resolution($`std_logic`$) =$ `resolved` defined as follows:

	’U’	’X’	’0’	’1’	’Z’	’W’	’L’	’H’	’-’
’U’	’U’	’U’	’U’	’U’	’U’	’U’	’U’	’U’	’U’
’X’	’U’	’X’	’X’	’X’	’X’	’X’	’X’	’X’	’X’
’0’	’U’	’X’	’0’	’X’	’0’	’0’	’0’	’0’	’X’
’1’	’U’	’X’	’X’	’1’	’1’	’1’	’1’	’1’	’X’
’Z’	’U’	’X’	’0’	’1’	’Z’	’W’	’L’	’H’	’X’
’W’	’U’	’X’	’0’	’1’	’W’	’W’	’W’	’W’	’X’
’L’	’U’	’X’	’0’	’1’	’L’	’W’	’L’	’W’	’X’
’H’	’U’	’X’	’0’	’1’	’H’	’W’	’W’	’H’	’X’
’-’	’U’	’X’	’X’	’X’	’X’	’X’	’X’	’X’	’X’

VHDL Type `std_logic_vector` Because of syntactical restrictions we only can define signals and variables of this type by giving the size of the vector. This is assumed to be done using `std_logic_vector ( 1 to n )` where n denotes the fixed size. Every element of the vector has the type `std_logic`. All operations defined for `std_logic` are assumed to exist for `std_logic_vector`. They are defined by successively applying the operation to the related vector elements. Binary operations are only allowed if the size of both involving vectors is the same. Access to only one vector element is done by the expression $I(E)$ where I must be a variable or signal of type `std_logic_vector` and E an expression.

2.2.4 VHDL Example Evaluations

We show the correctness of the proposed semantics by examples. Consider the following program implementing a simple signal assignment

```
S <= '0', '1' after 10ns, '0' after 20ns;
```

converted to the **RVHDL** syntax.

```
entity test is
```

```
end test;
architecture behavior1 of test is
     signal S: std_logic;
begin
     beh:  process
     begin
          S <= '0' after 0ns;
          S <= transport '1' after 10ns;
          S <= transport '0' after 20ns;
          wait until false;
     end process beh;
end behavior1;
```

From the program the following informations can be derived:

$SIG = \{S\}$

$PROCESS = \{beh\}$

$VARS = \{\}$

$DRIVER = \{S_{beh}\}$

$FUNC = \{resolved\}$ Note that *resolved* denotes the resolution function of `std_logic`.

Program execution starts with the initialization phase.

$ActualTime = 0ns$

$PActive(beh) = true$

$SActive(S) = true$, $SEvent(S) = true$, $SEnv(S) =' U'$

$Pow(S_{beh}) = \{\}$

At the end of the initialization phase the process *beh* is executed. Because the process only contains signal assignments, the associated projected output waveform is changed.

After the first statement, $Pow(S_{beh}) = \{('0', 0)\}$, after the second, $Pow(S_{beh}) = \{('0', 0), ('1', 10)\}$, and finally, $Pow(S_{beh}) = \{('0', 0), ('1', 10), ('0', 20)\}$.

The *wait* statement at the end of the process suspends the process. The initialization phase ends and the first simulation cycle starts. Since $('-', 0) \in Pow(S_{beh})$, the driver becomes active and the recomputation of the signal value for S starts. This leads to a new value on S which becomes active and $PEvent$ switches to true. No process is executed because S is not in any process's sensitivity list. The first simulation cycle stops. Since $('1', 10)$ is the first element of $Pow(S_{beh})$ at the beginning of the second cycle and no process become active, time advances to $10ns$ and the signal value of S is set to $'1'$. In the third cycle, time advances to $20ns$ and the value of S is changed to $'0'$. Because no other elements are inside $Pow(S)$, no further simulation cycle is invoked.

Chapter 3

Using Functional Dependencies for Software Debugging

3. Das logische Bild der Tatsachen ist der Gedanke.

Ludwig Wittgenstein, 1921
Tractatus logico-philosophicus
Logisch-philosophische Abhandlung

3.1 Motivation

In this chapter we describe the usage of functional dependencies for diagnosing very large VHDL programs as employed during the DDV project sponsored by Siemens Austria with the aim of applying model-based diagnosis to the domain of software debugging. Before introducing basic concepts, program conversions, and results we motivate the usage of automatic debugging tools.

Figure 3.1 is an overview of the hardware design cycle using a language such as VHDL (Very High Speed Integrated Circuit Hardware Description Language). A specification (written in VHDL) delineates the functional requirements for the hardware to be designed. This is followed by a detailed design on the register-transfer (RT) level, which, once it has passed all tests, is automatically converted ("synthesized") into a gate-level representation. The design is tested by running simulations of the VHDL code for a number of test cases. The simulations produce waveform traces of the values occurring over time on the signal lines in the simulated hardware (see Figure 3.2 for an example). The waveform traces of the program to be tested, usually comprising several 10,000 signal changes, are compared to the traces generated by the specification on the same input values. A discrepancy indicates the existence of a fault in the design. The average simulation runtime increases strongly throughout the design cycle as the design becomes more complex and detailed.

Large hardware designs (comprising multiple ASIC's and microprocessors) can reach dimensions

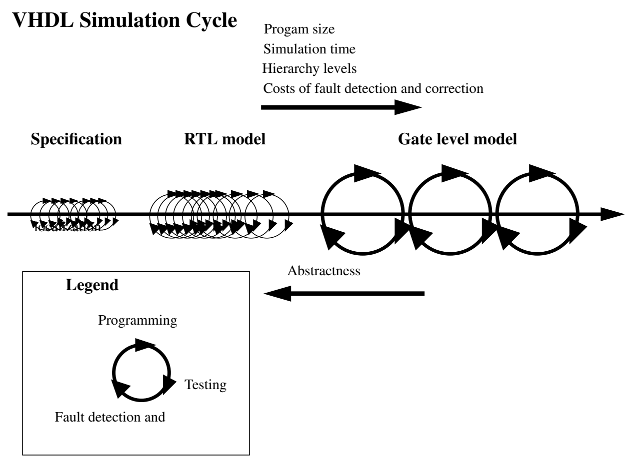


Figure 3.1: The design cycle using VHDL

of several 100.000 lines of VHDL code and thousands of components and signals at the top level. For such designs, typically written by a large team of designers, fault detection and localization becomes a very time-consuming activity.

However, given that even an individual simulation run takes from hours to days of real time on a high-end workstation, diagnosing a complete logical representation of the full VHDL program and its semantics is not feasible. A strongly abstracted view of the design must be used for diagnosis. We abstract over values and time points [Ham91], but retain the capability to distinguish between the initialization phase and operating mode of a circuit, a requirement for handling feedback loops. Sufficient discrimination between diagnoses is achieved by applying multiple test cases and measurement selection. Further discrimination can be achieved by requesting the user to evaluate the correctness of particular signals.

The diagnosis system as described in this part of the thesis was implemented to fit into the existing VHDL design environment and is intended for use in all design phases where VHDL is used.

The rest of this chapter is structured as follows: Section 3.2 gives an introduction to functional dependencies. Section 3.3 introduces the example we will use throughout the paper and defines the basic terminology required to apply model-based diagnosis to VHDL. In Section 3.4, we discuss the abstracted knowledge representation used in the tool and its complexity. Section 3.5 introduces an improved system description reducing the time complexity to a feasable level. This is followed by an overview of the implementation and test results of the diagnosis system (Section 3.8).

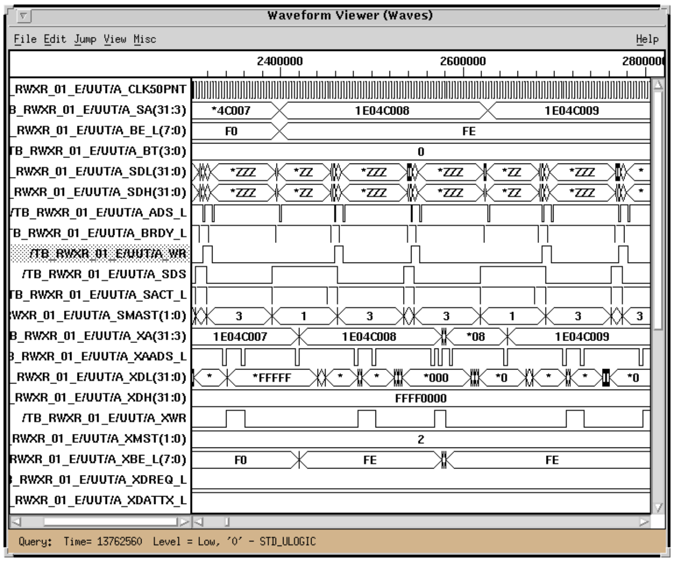


Figure 3.2: A typical waveform trace

3.2 Introducing Functional Dependencies

Functional dependencies (FDs) are introduced for describing relationships between signals and variables used in programs, statements and functions. Several papers like [Jac95] indepentently introduced quide similar concepts used in a different application domain. While Jackson's *Aspect* system ([Jac95]) uses dependencies between variables to check program correctness on an abstract level, we use them for diagnosis. Within the *Aspect* system dependencies between variables must be specified by the user. Programs are annotated with the intended dependencies and *Aspect* checks whether the program fulfills them or not.

We first introduce the concept of FDs and secondly show how they can be used for describing behavior dependencies between signals and variables of VHDL programs.

Informally a functional dependency between two dynamic objects (signals or variables) can be derived. If a change of the second object's value may cause a change of the first one.

Definition 3.2.1 (Functional Dependency) *Let OBJ be a set of constants denoting variables and signals, and Π be a program using the elements of OBJ. An object t depends functionally on an object s iff a value change on s may cause the program to compute a value change on t. A functional dependency is written as a pair (t, s).*

Consider for example the statement `MEM <= encrypt(DATA)`, which causes the FD $(MEM, DATA)$. Note that the `encrypt` function need not to compute a value changing the original value of MEM. Statements must not have only one functional dependency. Instead every statement can be involved in a set of functional dependencies. To compute such functional dependency sets we need operations defined on them. $\cup$ builds the union of two functional dependency sets. The operations $\circ$, $*$, and Π used to combine the functional dependency sets of concurrent and sequential statements are defined as follows.

Definition 3.2.2 *Let FD_1, FD_2 be two functional dependency sets, and S a subset of OBJ. The operations $\circ$, $*$, and Π are defined as follows:*

$FD_1 \circ FD_2 = \{(t,s)|(t,x) \in FD_2 \wedge (x,s) \in FD_1\} \cup FD_1$

FD_1* denotes the closure of successively applying $\circ$ on FD_1.

$\Pi_S FD_1 = \{(t,s)|(t,s) \in (FD_1*) \wedge t \in S \wedge s \in S\}$

Consider for example the sequential program part

```
...
St1:  a <= b ;
St2:  c <= d and a ;
...
```

The functional dependencies for statement `St1` are $FD(St1) = \{(a,b)\}$ and for `St2` $FD(St2) = \{(c,d),(c,a)\}$. The functional dependency set for "St1 ; St2;" then is given by

$FD(St1;St2;) = \{(c,d),(c,b),(a,b)\}$.

In general the functional dependencies for statement blocks can not be computed out of the functional dependencies of each individual statement. Consider for example the sequential program

```
...
St1:  a <= a - b ;
St2:  a <= a + b ;
...
```

used as an example in [Jac95]. In this case the variable `b` has no influence on the value of `a` after executing the statement `St2`. Only the dependency $\{(a,a)\}$ should be derived.

Between the functional dependencies of a sequence of statements and each individual statement the following connection can be drawn:

$FD(St1;St2;) \subseteq FD(St1) \circ FD(St2)$

The exact functional dependency set for the sequence can only computed using semantics information. However, computation of functional dependencies should be a non time consuming activity and it is not very likely that a program contains the above statements. Therefore, in the rest of this thesis we only use the equation

$FD(St1;St2;) = FD(St1) \circ FD(St2)$

to compute the functional dependency set for statement blocks. This may lead to the computation of too may dependencies, e.g., in the above example we get $\{(a,a),(a,b)\}$ as result.

To express that a signal s depends on a set of other signals we introduce the notation $s \leftarrow_{FD} \{s_1,\ldots,s_n\}$.

In the next subsection we show how functional dependencies can be computed out of VHDL programs, i.e., we define how to compute the FD function associating statements (concurrent and sequential) with functional dependency sets.

3.2.1 Computing Functional Dependencies

In VHDL a program is divided into entities with associated architectures. An entity can be accessed via its identifier, and an architecture via its entity and its identifier. Since entities have an interface, communication between entities can only be done using the signals of the interface. Therefore we are only interested in describing functional dependencies using signals defined in the interface. On the other hand we can compute a functional dependency set using signals of the interface and signals defined in the architecture. In the following we assume that E denotes an entity and A it's associated architecture.

$FD(E) = \Pi_{SIG(E)} FD(A)$

SIG is a function returning the signals defined in an object.

The interface signals for entity E are given by

$SIG(E) = SIG(\texttt{entity } A \texttt{ is begin port(}I\texttt{); end ;}) = SIG(I)$

and if no interface is defined

$SIG(E) = SIG(\texttt{entity } A \texttt{ is begin end ;}) = \{\}$.

The signals can be computed from the interface declarations consisting out of $I_1,\ldots,I_n$ (where I_i denotes an interface declaration) as follows:

$SIG(I_1;\ldots,I_n) = \bigcup_{i=1}^{n} SIG(I_i)$

and

$SIG(I_i) = SIG(\texttt{signal}IL\ :\ldots) = SIG(IL)$

where IL denotes an identifier list. If the identifier list consists of identifiers $Id_1,\ldots,Id_m$, then $SIG(Id_1,\ldots,Id_m) = \{Id_1,\ldots,Id_m\}$.

Note that we do not prove if the signal usage matches its signal interface declaration, e.g., if a signal is declared as *in signal*, then it is not allowed to be used as target of a signal assignment used in the associated architecture.

The functional dependency set for architectures is given by:

$$FD(A) = FD(\texttt{architecture } A \texttt{ of } E \texttt{ is } AD \texttt{ begin } CS \texttt{ end ;}) = \\ \Pi_{(SIG(A)\cup SIG(E))} FD(CS)$$

where AD denotes architecture declarations and CS the concurrent statements. Signals defined in the architecture declaration can be computed as before.

$SIG(A) = SIG(AD)$

Let $Sd_1; \ldots Sd_m;$ be the sequence of signal declarations.

$SIG(Sd_m; \ldots, Sd_m) = \bigcup_{i=1}^{m} SIG(Sd_m)$

and

$SIG(Sd_i) = SIG(\texttt{signal } IL \; : \ldots) = SIG(IL)$

where IL denotes an identifier list.

Since concurrent statements work in parallel, several statements can change a signal's value at the same time. To avoid problems regarding value update, a resolution function determines the exact signal value using a driving value. In VHDL there are implicit signals, one for each driver. The driving value is then defined as the value of the driver. While computing a functional dependency set out of concurrent signals we do not take account of drivers. Let $C_1, \ldots, C_m$ be concurrent statements used in architecture A. The overall functional dependency is given by:

$FD(CS) = \bigcup_{i=1} mFD(C_i)$.

The functional dependencies for concurrent statements can be computed by:

$FD(C_i) = FD(PI : \texttt{ process } PD \texttt{ begin } SQ \; WS \texttt{ end process } PI \;;) = (FD(SQ) \cup (out(FD(SQ)) \times \Pi_{(SIG(E) \cup SIG(A) \cup VAR(PI))} SIG(WS))$

where PI is the name of the process, SQ the sequence of statements, WS the wait statement, and PD the process delarative part. Since wait statements influence the time behavior, this must be represented in the functional dependency set. If a signal change causes the execution of the wait, then another signal can change its value. This holds only for signals used as target. Therefore we construct new dependencies by using target signals and signals included in the wait statement. SD may contain variable declarations. The function VAR answers a set of variables declared in the process.

$VAR(PI) = VAR(PD)$

Assume that the declarative part is given by $PD = V_1; \ldots, V_n;$.

$VAR(V_1; \ldots V_n;) = \bigcup_{i=1} nVAR(V_i)$

where

$VAR(V_i) = VAR(\texttt{variable } IL\texttt{:} \ldots) = VAR(IL)$

IL denotes an identifier list and obviously $VAR(IL) = SIG(IL)$ holds. As introduced in the previous section, functional dependencies of sequential statements can be computed using the $\circ$ operator. Let $SQ = St_1; \ldots St_n$ be the statement part of process PI.

$FD(SQ) = FD(St_1; \ldots St_n;) = FD(St_1) \circ FD(St_2; \ldots St_n)$

The functional dependency sets for the single sequential statements are given as follows:

- *Signal Assignment* The signal written on the left side of the signal assignment (target signal) depends functionally on every signal and variable used on the right side. In practical VHDL programs time expressions TE not use variables or signals and are therefore not considered in this context. The functional dependency set is therefore given by:

  $FD(T\texttt{<=}\ E\ \texttt{after}\ TE) = \{(T,x)|x \in SIG(E) \vee x \in VAR(E)\}$

  T denotes the target signal, E the expression, and TE the time expression. In the case where a transport keyword is used the computation of functional dependencies does not change.

  $FD(T\texttt{<=}\ \texttt{transport}E\ \texttt{after}\ TE) = \{(T,x)|x \in SIG(E) \vee x \in VAR(E)\}$

- *Variable Assignment* The functional dependencies for variable assignment statements are computed as for signal assignment statements.The target variable depends functionally on every signal and variable used on the right side. The functional dependency set is therefore given by:

  $FD(T\ \texttt{:=}\ E) = \{(T,x)|x \in SIG(E) \vee x \in VAR(E)\}$

  T denotes the target signal and E the expression possible changing the value of T.

- *Conditional Statement* Because we do not know the execution path of a conditional without evaluating the condition, we must include the functional dependencies of both statement parts in the functional dependency set of the conditional statement. In addition, the signals in the condition influence the output signals of the statement parts.

  $FD(\texttt{if}\ E\ \texttt{then}\ SEQ_1\ \texttt{else}\ SEQ_2\ \texttt{end if}) =$

  $$FD(SEQ_1) \cup FD(SEQ_2) \cup ((out(FD(SEQ_1)) \cup out(FD(SEQ_2))) \times SIG(E))$$

  where E denotes the condition, SEQ_1 the statement part of the *then* and SEQ_2 the statement part of the *else* branch.

- *Loop Statement* With similar arguments as given above for conditionals we also must include signals used in the condition in the functional dependency set. Consider for example the following program part:

```
...
I := 0;
while I < 8 loop
     S(I) <= I + 5;
     I := I + 1;
end loop;
...
```

  $FD(\texttt{while}\ E\ \texttt{loop}\ SEQ\ \texttt{end loop}) = FD(SEQ) \cup (out(FD(SEQ)) \times SIG(E))$ where E is the condition, and SEQ the statement part of the loop.

- *Null Statement* The null statement influence no signal or variable values. The FD function therefore returns an empty set.

  $FD(\texttt{null}) = \{\}$

- *Wait Statement* Wait statements only influence the timing behavior of the process. Variable and signal values are not changed directly. FD must therefore answer the empty set.

  $FD($`wait ...`$) = \{\}$

  On the other hand, the wait statement has an implicite effect on variable and signal values. Consider for example the process

  ```
  ...
  S2 <= S1 and S0 ;
  wait on S1;
  ...
  ```

  where the signal `S0` has forgotten to be included into the waits signal list. In this case a change on `S0` does not change the value of `S2`.

  To express this implicit influence we use the wait statements signals which are computed as follows:

  $SIG($`wait on` $IL) = SIG(IL)$ where IL denotes an identifier list.

  $SIG($`wait for` $T) = \{\}$ where T denotes a time expression.

  $SIG($`wait until` $E) = SIG(E)$ where E denotes a condition which must be true to execute the statement.

Signals and variables must be computed from expressions using the SIG respective VAR function. We describe only the definition of the SIG function, the definition of VAR is analogous.

1. *Expressions*

   $SIG(R_1\ Op\ R_2) = SIG(R_1) \cup SIG(R_2)$ where R_i are relations and Op is a logic operation (`and` , `or` , `xor` , `nand` , `nor`).

2. *Relations*

   $SIG(SE_1\ Op\ SE_2) = SIG(SE_1) \cup SIG(SE_2)$ where SE_i are simple expressions and Op is a relation operation (`=` , `/=` , `<`, `<=`, `>`, `>=`).

3. *Simple Expressions*

   $SIG(Si\ T_1) = SIG(T)$

   $SIG(Si\ T_1\ Op\ T_2) = SIG(T_1) \cup SIG(T_2)$ where Si denotes the signum (`+`, `-`).

   $SIG(T_1) = SIG(T)$

   $SIG(T_1\ Op\ T_2) = SIG(T_1) \cup SIG(T_2)$

   where T_i is a term and Op an adding operator (`+`, `-` , `&`).

4. *Terms*

   $SIG(F_1\ Op\ F_2)$ where F_i are factors and Op a multiplication operator (`*`, `/`, `mod`, `rem`).

5. *Factors*

   $SIG(P_1 \;{*}{*}\; P_2) = SIG(P_1) \cup SIG(P_2)$

   $SIG(\text{abs } P_1) = SIG(P_1)$

   $SIG(F) = SIG(\text{not } P_1) = SIG(P_1)$

   where P_i denotes primaries.

6. *Primaries*

   $SIG(I) = \{I\}$ where I denotes an identifier.

   $SIG(I\ PR) = \{I\} \cup SIG(PR)$ if I denotes a signal storing an array, and PR the rest of the primary.

   $SIG(I\ PR) = SIG(PR)$ if I denotes a function, and PR the rest of the primary.

   $SIG(L) = \{\}$ where L denotes a literal, e.g., '1'.

   $SIG((\ E\ )) = SIG(E)$ where E denotes an expression.

   $SIG((\ E_1, \ldots, E_n\ )) = \bigcup_{i=1}^{n} SIG(E_i)$ where E_i denotes an expression

This finishes the definition of the FD, SIG and VAR function, and we can compute functional dependencies for entities, architectures, and statements. The following example gives the functional dependencies for a VHDL program.

Example 4: Consider the running example 1.1. The functional dependencies are given for each program part in boldface format.

```
entity MEM is
-- FD(MEM) = {(data,data), (data,rw), (data,clk),(data,de)}
     port (
-- SIG = { clk, de, rw, data }
          signal clk, de, rw :  in std_logic;
          signal data :  inout std_logic );
end MEM;
architecture BEHAV of MEM is
-- FD(BEHAV) = {(data,data), (data,rw), (data,clk), (data,de)}
begin
     MEM_UNIT: process
-- FD(MEM_UNIT) = {(data,data), (data,rw), (data,clk), (data,de) }
          variable mem:  std_logic;
-- SIG = { mem }
     begin
          if clk = '1' and de = '1' then
-- FD = {(mem,data),(data,mem), (mem,rw), (data,rw), (mem,clk), (mem,de), (data,clk), (data,de) }
               if rw = '1' then
-- FD = {(mem,data),(data,mem), (mem,rw), (data,rw) }
                    mem := data;
-- FD = { (mem,data) }
```

```
          else
               data <= mem;
-- FD = { (data,mem) }
          end if;
     end if;
     wait on clk;
  end process;
end BEHAV;
```

Now it can be seen that data depends on all other signals, i.e., $data \leftarrow_{FD} \{data, clk, rw, de\}$ which is exactly the expected dependency. □

Note that arrays are not specially treated in the current definitions. This may lead to unsatisfactory results in diagnosis. Consider for example the following program part, where we assume that statement St1 is wrong because it should refer to B.

```
...
St1:  process
begin
     A(0) <= not B ;
end process St1;
St2:  process
begin
     A(1) <= not B ;
end process St2;
...
```

In the current definition, only the functional dependency set $\{(A, B)\}$ holds for St1 and St2, which does not allow to differentiate between them while diagnosing the program.

To avoid such problems we must redefine the SIG and VAR function, taking into account that every subelement of an array is itself a single signal. However, this causes a new problem.

If the subelement of the array cannot be determined statically, i.e., at compile time, the computation of functional dependencies for the subelements is not possible. We therefore treat subelements in the improved computation of SIG as a seperate signal, if the index can be determined at compile time, i.e., the index is a literal.

To do this we change the computation of $I\ PR$ for the SIG function (see item 6) such as follows:

$SIG(P) = SIG(I\ PR) = SIG(PR)$ where I denotes a function call, and PR the rest of the primary.

$SIG(P) = SIG(I\ PR) = \{I\} \cup SIG(PR)$ where I denotes an array where the index given by PR is not a literal, and PR the rest of the primary.

$SIG(P) = SIG(I\ PR) = \{I(value(PR))\}$ where I denotes a an array where the index given by

Figure 3.3: The D75 - Diagnosis components

PR is a literal, and PR the rest of the primary. It is assumed that the literal can be accessed via a *value* function (e.g., $eval_E$).

This completes the computation of functional dependencies out of **RVHDL** programs.

3.3 A very short VHDL example

To give an example of how to design models and find faults in VHDL, we introduce a simple VHDL program which represents a device we have called D75. The D75 is a sequential device with five input ports a, b, c, d, e, three outputs f, g, h and the following functional specification.

$$
\begin{aligned}
&f = \begin{cases} state_t + a * b & \text{if } b > 0 \\ state_t & otherwise \end{cases} \\
&g = \begin{cases} -b * c & \text{if } b \leq 0 \\ 0 & otherwise \end{cases} \\
&h = state_t \\
&\text{where} \\
&state_t = \begin{cases} (d + e + state_{t-1}) \text{ after 5 ns}^{1} & \text{if } t > 0 \\ 0 & otherwise \end{cases}
\end{aligned}
$$

This formal specification can be easily transformed to a specification written in VHDL, which allows us to simulate the specification and produce a waveform trace.

Figure 3.3 shows a possible design of the D75 consisting of four components. A designer who uses VHDL must now implement a program for every component to get the same behavior as the specification. The behavior can be tested by checking the waveform traces for every test

[1] The after keyword indicates that the new value is assigned with a delay of 5 ns.

case specified by the designer with the specification traces. If this check leads to discrepancies we want to find the component that causes the misbehavior. The dotted lines in Figure 3.3 show the functional dependencies between inputs and outputs of the components.

The following VHDL code fragment describes the internal topology of the D75 as shown in figure 3.3.

```
entity d75_e is
    port( signal a,b,c,d,e: in integer;
          signal f,g,h: out integer);
end d75_e;


architecture a2 of d75_e is
    ...
    signal s1,s2,s3: integer := 0;
    ...
begin
    i1: c1 port map (a,b,c,s1,s2);
    i2: c2 port map (s1,s3,f);
    i3: c3 port map (d,e,s3,s3);
    i4: c4 port map (s2,s3,g,h);
end a2;
```

This VHDL top level implementation of the D75 shows the most important VHDL components, *entities* and *architectures*. Entities are interface declarations, whereas architectures are behavior definitions, i.e., they associate a behavior with an entity. An architecture consists of a set of *concurrent statements*, so called because they are executed in parallel (`i1` ... `i4` in the example above). Communication between concurrent statements is achieved by *signals*. Signals are defined in the declaration part of the architecture or entity, or in the interface declaration (port ...) of the entity. VHDL is strongly typed. For further details on VHDL syntax and semantics see, e.g., [VHD88], [Nav93], or [Bha92].

Definition 3.3.1 *Let A be an architecture and E an entity. $ConcSt(A, E)$ denotes the set of concurrent statements defined in A. $Signals(A, E)$ denotes the set of signals defined in A and E.*

Example:

$$ConcSt(a2, d75\_e) = \left\{ \begin{array}{l} \text{i1: c1 port map (a,b,c,s1,s2);} \\ \text{i2: c2 port map (s1,s3,f);} \\ \text{i3: c3 port map (d,e,s3,s3);} \\ \text{i4: c4 port map (s2,s3,g,h);} \end{array} \right\}$$

$Signals(a2, cc2\_e) = \{a, \ldots, h, s1, s2, s3\}$.

For simplicity we will write down only the label of the concurrent statement (e.g. $i1 \ldots i4$) instead of the whole statement in this thesis. $i1$ thus stands for "$i1 : c1$ *port map*$(a, b, c, s1, s2)$" and so on.

Until now we have not defined the behavior of the concurrent statements in the introduced D75 example. To do this we write down signal assignments that are supposed to describe the behavior for every concurrent statement $i1 \ldots i4$. We write `in`i for the i-th input and `out`i for the i-th output of the concurrent statement. We order the signals of the port maps so that input signals come first and outputs last.

- $i1$ port map(in1,in2,in3,out1,out2)

```
out1 <= in1 * in2;
out2 <= (-1)*(in2 * in3);
```

- $i2$ port map(in1,in2,out1)

```
out1 <= in1 + in2;
```

- $i3$ port map(in1,in2,in3,out1)

```
out1 <= in1 + in2 + in3 after 5 ns;
```

- $i4$ port map(in1,in2,out1,out2)

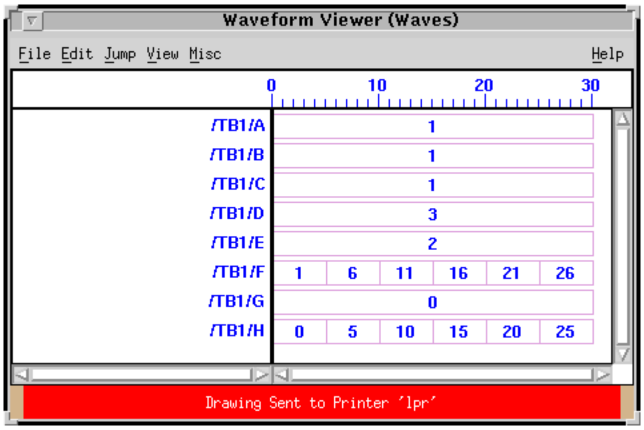
```
out1 <= in1;
out2 <= in2;
```

The VHDL program can now be tested. In the example, we have two test cases which stimulate the D75 with the following initial input vectors.

Signal	a	b	c	d	e
Test Case 1	1	1	1	3	2
Test Case 2	3	-1	1	4	2

The VHDL simulator produces the waveform trace for test case 2 shown in figure 3.4, which can be checked for correctness with regard to a specification or an older (correct) version of the program. In the D75 example we have a formal specification that can be transformed to VHDL. We can thus show the correctness of a VHDL program *for a given input pattern and a certain time interval* by comparing (automatically) the trace from the specification execution with the trace of the VHDL program. The discrepancies are then used as a starting point for computing diagnoses. Figure 3.4 shows the specification trace of the D75 example for both test cases.

More formally, a trace is defined as follows:



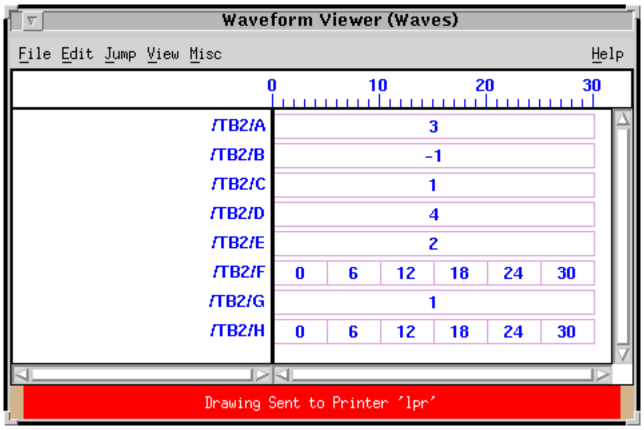


Figure 3.4: Specification Waveform Trace - Test Case 1 and 2 of the D75

Definition 3.3.2 (Waveform Trace) *Let A be an architecture and E an entity. The waveform trace $Trace(A, E)$ is a set of tuples (called* events*) (s, v, t) representing that a certain signal s (defined in A and E) changes to value v at time t.*
$Trace(A, E) = \{\langle s, v, t\rangle | s \in Signals(A, E) \wedge v \in Type(s) \wedge t \in Time\}$ *with*

1. *$Type(s)$ ... represents the domain of values for signal s.*
2. *$Time$ ... represents a discrete set of time values ≥ 0.*

The value of each signal at an arbitrary point in time can be observed from the waveform trace:

Definition 3.3.3 (Value) *Let $Trace(A, E)$ be a waveform trace for A and E. The value of each signal $s \in Signals(A, E)$ at time t, denoted by $Value(s, t)$, is the value that s has taken at the last event before t that changed s, i.e. $Value(s, t) = v_0$ if $\exists\langle s, v_0, t'\rangle \in Trace(A, E) \wedge t' \leq t \wedge \nexists\langle s, v_1, t''\rangle \in Trace(A, E) : t' < t'' \leq t \wedge v_0 \neq v_1$.*

Now we are able to formally define discrepancies.

Definition 3.3.4 (Discrepancy) *The discrepancies of A and E, written as $\mathcal{D}(A, E)$, are the set of tuples $\langle s, v, t\rangle$ where $s \in Signals(A, E)$, $t \in Time$, $v \in Type(s)$ is the expected value of the signal s, and $Value(s, t) \neq v$. We write $Discrepancy(s) \leftrightarrow \exists\langle s, v, t\rangle \in \mathcal{D}(A, E)$*

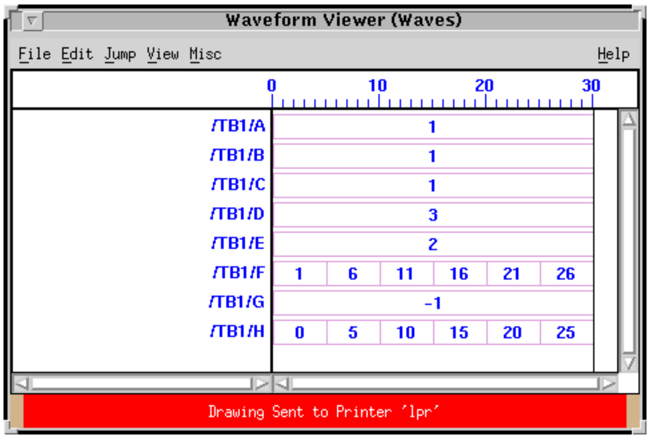
A signal discrepancy can be found automatically by comparing waveform traces or as result of an oracle, e.g., input by the designer of the VHDL program. A signal discrepancy and the description of the VHDL program (e.g., its concurrent statements) are the input for the diagnosis engine. Figure 3.5 shows the program waveform trace for test case 1 and 2, where discrepancies for signals g and f, respectively can be detected confirming the existence of a misbehavior.

3.4 Computing Diagnoses

In this section we formalize the diagnosis knowledge used in the VHDL-diagnosis system. The complexity of the VHDL programs to be diagnosed precludes analysis down to the last detail. The goal, rather, must be to compute results that will enable the designer to pinpoint the actual fault and find it faster.

Because of the complexity of the problem, we have to be content with a significantly abstracted representation, governed by these criteria:

- No diagnoses may be excluded due to abstractions.
- Integration with available commercial simulation packages.
- Computational costs must be minimized by requirering very few additional simulation runs.



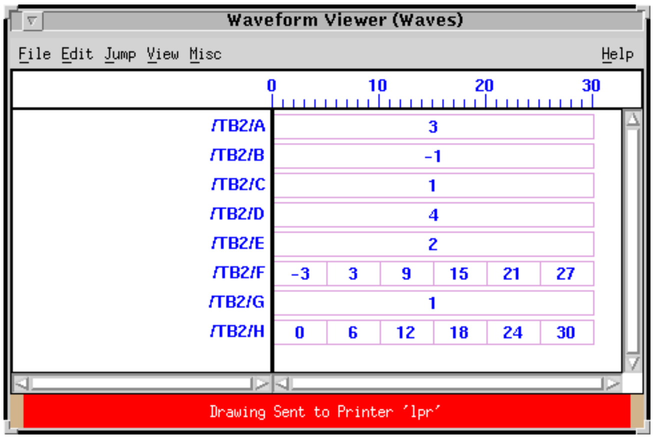


Figure 3.5: Waveform Trace - Test Case 1 and 2 of the D75

The principal idea is to abstract as much as possible over time and values on the one hand, while preserving the capability to discriminate between substantial parts of the VHDL-code on the other hand.

3.4.1 Components

VHDL code is partitioned into hierarchies of concurrent statements which usually bottom out in a sequence of sequential statements. Such a sequence contained in one concurrent statement typically comprises only a small fraction of the complete VHDL program, and as a first approximation it is not worth the effort needed to pinpoint individual sequential statements as the source of a fault. Therefore, identifying faulty concurrent statements is the main goal.

Definition 3.4.1 (Components) *The set of components $COMPS$ of an entity E and an associated architecture A consists of all concurrent statements of A; i.e. $COMPS(A, E) = ConcSt(A, E)$*

Example (cont.): In the D75 example, $COMPS(a2, D75) = \{i1, i2, i3, i4\}$.

Components are connected by signals which are defined in the VHDL code. In principle, VHDL distinguishes between in, out, and inout signals. Each component defines how it uses signals which means that we are able to specify three different sets of ports for a component.

Definition 3.4.2 (Input, Output, Input/output) *Let C be a component of entity E and architecture A. For each components C, the set of all signals that are used in C and declared as inputs, is called the set of input signals of C denoted by $in(C)$. The sets of output/inout signals, called $out(C)$ and $inout(C)$, are defined in the same manner.*

Example (cont.): $in(i1) = \{a, b, c\}$, $out(i1) = \{s_1, s_2\}$

For here on we represent each inout signal as a combination of one input and one output signal.

3.4.2 System description

The introduced system description uses functional dependencies as defined above. This has the big advantage that (1) the simulation of VHDL programs can be executed by commercial simulation tools, and (2) that the conversion of VHDL code to functional dependencies can be done statically.

After conversion functional dependencies (encoded in rules) are used to find the components which are responsible for the discrepancies.

Example (cont.): s_1 depends functionally on signals a and b ($s_1 \leftarrow_{FD} \{a, b\}$).

The main idea of the introduced abstraction is to ignore time and values of the signals. In the system description we only talk about correct or incorrect signals which means that we don't care when the discrepancy occurred and what the expected value would be. In particular, the output of a component is correct if the component is correct and the input signals (on which the output depends) are correct.

On one hand this abstraction has the advantage that we do not need to reason about thousands of time points for hundreds of signals. On the other hand it has the obvious drawback that we can exclude fewer diagnoses than with a detailed model. However, programmers use this abstraction during debugging very effectively to identify the faulty concurrent statement.

So far we have completely abstracted over time and signal values in order to gain an efficient diagnosis system. However, in order to handle feedback cycles and the initialization phase of the circuit, at least a simple representation of time must be included in the model.

Definition 3.4.3 (SD part 1: components) *Let A be an architecture and E an entity. The system description modeling the component behavior of the VHDL program given by E and A is a set of sentences for every component $C \in COMPS$ and $out_i \in out(C)$ of the form*

$ok(C) \wedge ok(in_1)_T \wedge \ldots \wedge ok(in_n)_T \rightarrow ok(C, out_i)_{T+1}$

and for the initialization

$ok(C) \rightarrow ok(C, out_i)_0$

where out_i depends functionally on the input signals $in_1 \ldots in_n$ ($out_i \leftarrow_{FD} \{in_1, \ldots, in_n\}$).

The correct behavior of a single VHDL concurrent statement is described by multiple rules. The running example results in a separate rule for each signal assignment:

$ok(i1) \wedge ok(a)_T \wedge ok(b)_T \rightarrow ok(i1, s1)_{T+1}$
$ok(i1) \wedge ok(b)_T \wedge ok(c)_T \rightarrow ok(i1, s2)_{T+1}$
$ok(i2) \wedge ok(s1)_T \wedge ok(s3)_T \rightarrow ok(i2, f)_{T+1}$
$ok(i3) \wedge ok(d)_T \wedge ok(e)_T \wedge ok(s3)_T \rightarrow ok(i3, s3)_{T+1}$
$ok(i4) \wedge ok(s2)_T \rightarrow ok(i4, g)_{T+1}$
$ok(i4) \wedge ok(s3)_T \rightarrow ok(i4, h)_{T+1}$

The same signal can be declared as output in different concurrent statements. In this case a simulator uses a special *resolution function* to deduce the final value of a signal at a given time point. This value depends on all the components which use the signal as output.

Definition 3.4.4 (SD part 2: connections) *Let A be an architecture, E an entity, and $Driver(s) = \{c | c \in COMPS(A, E) \wedge s \in out(c)\}$ where s is a signal.*

The system description modeling the connections of the VHDL program given by E and A is the set of sentences which includes for each signal $s \in Signals(A, E) \wedge s \in out(c) \wedge c \in COMPS$ the sentence

$\bigwedge_{c_j \in Driver(s)} ok(c_j, s)_T \rightarrow ok(s)_T$

Example: Given component $i3$ from the D75 example the rule

$ok(i3, s3)_T \rightarrow ok(s3)_T$

must be element of the system description.

3.4.3 Observations

After the programming phase, tests, i.e., simulation runs, are conducted, possibly leading to the discovery of a misbehavior of the system. Misbehavior is detected by comparing selected outputs of the program with the outputs of the specification. We will call these selected outputs together with the stimulated inputs *traced signals*.

Definition 3.4.5 (Traced Signals) *The set of traced signals $TSIG$ consists of all signals that are defined in the entity, associated architecture and libraries and that are included in the wave-*

form trace.
$TSIG = \{s | s \in Signals(A, E) \wedge \langle s, v, t\rangle \in Trace(A, E)\}$

Example (cont.): $TSIG = \{a, \ldots, h, \}$

The traces of the signals in $TSIG$ are used to deduce observations in the sense of consistency-based diagnosis. If there are no discrepancies in the trace for a signal s, the $ok(s)$ literal is element of the set of observations. If a discrepancy has been found, then $\neg ok(s) \in OBS$.

Signals that are not element of the trace are not observed, but may be included in an additional simulation, depending on their information content and their observability. A signal is observable if either a specification for this signal exists or the user is willing to perform manual inspection.

Definition 3.4.6 (Observation) *Let $TSIG$ be a set of traced signals. The set of observations OBS is defined as*

$$\begin{aligned} OBS = \; & \{\neg ok(s) | s \in TSIG \wedge Discrepancy(s)\} \cup \\ & \{ok(s) | s \in TSIG \wedge \neg Discrepancy(s)\} \end{aligned}$$

Example (cont.):

Test Case 1:
$OBS = \{ok(a), \ldots, ok(f), \neg ok(g), ok(h)\}$

Test Case 2:
$OBS = \{ok(a), \ldots, ok(e), \neg ok(f), ok(g), ok(h)\}$

Next, we need to assign observations to particular points in time since we have associated time points to the $ok/1$ predicate declaring a signal to be correct. Since we ignore the exact time point when a signal becomes incorrect we have chosen an assignment of the observation to a given time point such that no diagnoses are excluded. Therefore every conflict in computed using the abstraction has to be a superset of (or equal to) a conflict in a "hypothetical most accurate model". In other words, we have to make sure that conflicts are not too small.

In the rest of this section we assume that all exogenous quantities, e.g., external input signals, are assumed to be correct. This removes the need to reason about incorrect behavior of the outside world. Since we do not consider individual signal values, it follows from this assumption that it is sufficient to restrict the set of signals represented in the system description to those that actually are outputs for a component.

In the following, for a given pair of vertices v and v', such that a path from s to s' exists in the graph, we use the term *component distance* to denote the number of intermediate components on the shortest such path.

Example: If the shortest path between signals s and s' is given by $< s, c_1, s_1, c_2, s' >$, the component distance is 2. If the the shortest path between components c and c' is given by $< c, s_1, c_1, s_2, c' >$, the component distance is 1.

Lemma 3.4.1 *Let A be an architecture and E an entity, and let $s, s' \in Signals(A, E)$ such that the component distance from s to s' is d. If $ok(s)_t$ does not hold for some time $t \in \{1, \ldots, n\}$, then $ok(s')_{t+d}$ cannot be derived.*

Proof: Note that for the correctness of an output signal to be derived at time t, all components that assign to the signal must behave correctly, and all input signals of these components must be correct at time $t-1$. Now, let $< s, c_1, s_1, \ldots, s_{d-1}, c_d, s' >$ be the shortest path from s to s'. Since $ok(s)_t$ does not hold, c_1 cannot produce a correct value on its outputs at time $t+1$, i.e., $ok((c_1, s_1))_{t+1}$ does not hold, nor does $ok(s_1)_{t+1}$. In general, for any signal s_i on the path, $ok(s_i)'_t$ will be derived for $t \leq i-1$. At time $d-1$, the "missing" ok value of s reaches s_{d-1}, therefore $ok((c_d, s'))_{t+d}$ cannot be derived, therefore $ok(s')_{t+d}$ cannot be derived. ■

Corollary 3.4.2 *Let A be an architecture and E an entity, and let $c \in COMP$, $s \in Signals(A, E)$ such that the component distance from c to s' is d. If $ok(c)$ does not hold, then $ok(s')_t$ cannot be derived for $t \geq d$.*

Proof: Since c is not ok, the output signals of c will not be ok at time 0 (Def 3.4, initialization). If the distance from c to s is d, then according to Lemma 3.4.1, $ok(s)_t$ will be derived until time $d-1$, but not for $t = d$. Since the output signals of c will not be ok after time 0, either, $ok(s)_t$ will not be derived for any $t \geq d$ as well. ■

Lemma 3.4.3 *Given an architecture A and an entity E, let s be a signal and let $C = \{c_1, \ldots, c_m\}$ be the set of all components in A for which a path exists from c_i to s (i.e., s is functionally dependent on c_i). Because of the structure of the system description, if $ok(c_i)$ holds for every $i \in \{1, \ldots, m\}$, then $ok(s)_t$ holds for $t \geq 0$ (i.e., s is always correct).*

Proof: The proof is by induction over time. For $t = 0$, for every component c and output signal s' of c it follows from Definition 3.4 that $ok((c, s'))_0$, and from Definition 3.5 that $ok(s')_0$. This trivially includes all those output signals s' that actually lie on the path from c_i to s.

For $t > 0$, the induction hypothesis is that for all signals s' that lie on a path from $c \in C$ to s, $ok(s')_{t-1}$ holds. From this, we conclude by Def. 3.4 that $ok((c, out_i))_t$ holds for all components c and output signals out_i, and therefore by Def. 3.5 that $ok(s)_t$ holds for all signals s. ■

Lemma 3.4.4 *Let A be an architecture and E an entity. Because of the structure of the system description we are able to conclude that for $n = |COMPS|$ and each signal $s \in Signals(A, E)$, the value of s is ok for any time $t \geq n$ iff it is ok at time n:*

$\forall S \in SIGS, \forall t \geq |COMPS| : ok(S)_{|COMPS|} \leftrightarrow ok(S)_t$.

Proof:

Case 1. Assume that all components in the system are ok ($\forall c \in COMPS : ok(c)$). This means that for every signal s, all components that s depends on are ok. Therefore, it follows from Lemma 3.4.3 that for all signals, $ok(s)_t$ holds for $t \geq 0$ (i.e., all signals are correct forever, which is only to be expected). This proves Case 1.

Case 2. Assume that there is at least one component in the system that is not known to be ok. We have to distinguish between signals that are functionally dependent upon such a component and signals that depend only on components that are ok. For a signal s that depends solely on ok components[2], we can conclude as above that the signal will be ok for all time points $t \geq 0$.

Now consider a signal s that depends on at least one component c for which $ok(c)$ is not known to hold (without limitation of generality, if there are several, take c to be the one closest to s). From Corollary 3.4.2, we know that $ok(s)_t$ cannot hold for $t \geq d$, where d is the distance from c to s. Since there are n components in the system and c cannot be on the path, it must hold that $d < n$. This proves the lemma. ■

Because of lemma 3.4.4, it is sufficient to consider n time points where n is the number of components and to assert $\neg ok(s) \leftrightarrow \neg ok(s)_n$ for each $s \in TSIG \wedge Discrepancy(s)$.

For each $s \in TSIG \wedge \neg Discrepancy(s)$ we have to assert $ok(s) \leftrightarrow \bigwedge_{i=0 \dots n} ok(s)_i$

Example (cont.):

Let us consider test case 2 and only signals f and g. We have to assert

$ok(f) \leftrightarrow ok(f)_4 \qquad ok(g) \leftrightarrow \bigwedge_{i=0 \dots 4} ok(g)_i$.

3.4.4 Diagnosis

For the definition of diagnoses we apply the consistency-based view as introduced in chapter 1.2.1.

Since in the application domain in almost all cases predefined tests are applied to a VHDL program, we incorporate tests in the definition of diagnosis. Note that because of the used abstraction we do not need to transfer knowledge across tests as in [RdKSS91]. Diagnosis with tests can be defined as follows:

Definition 3.4.7 (Diagnosis wrt Tests) *Let $TOBS$ be a set of OBS_i where each OBS_i represents the observations for a particular test.*

A mode assignment Δ is a diagnosis for a VHDL program $\langle A, E \rangle$ iff for each $OBS_i \in TOBS$

$SD \cup OBS_i \cup \Delta \cup \{ok(c) | c \in COMP \setminus \gamma(\Delta)\}$

is consistent.

[2]Note that functional dependency is transitive - this means there may be no path of whatever length traceable to s from a component that is not ok

Consider again the D75 example. For test case 1 there exist two minimal single fault diagnoses: $[i1]$ and $[i4]$ (the conflict is $\langle i1, i4\rangle$). For test case 2 there exist two minimal single fault diagnoses: $[i1]$, $[i2]$ and $[i3]$ (the conflict is $\langle i1, i2, i3\rangle$).

Therefore, there is one minimal single fault diagnosis $[i1]$. Inspecting the definition of $i1$, we note that the if statement concerning the value of signal b is missing. As a result, we achieved a fault localization with two test cases.

As noted, we designed the knowledge representation according to the safety principle that no diagnosis may be excluded. On one hand, this may result in too many diagnoses. However, this is not a serious problem if multiple test cases are applied, and in addition the user can be asked to judge the correctness of a signal.

On the other hand, the calculation of diagnoses is simple and fast enough for computing diagnoses even for large programs. This is exactly the case where assistance for fault localization has a very high return on investment.

Note that if we were willing to accept lower performance for the sake of more detailed reasoning about the VHDL program, we still would need to cope with a modeling problem since often physical hardware components (like microprocessors) are used to speed up the simulation. For such components a VHDL model or some other formal description would have to be specified, which would increase the design development costs significantly.

3.4.5 Complexity issues

Unfortunately the need to introducing discrete time points implies that the size of the system description does not increase linearly with the size of the system.

Theorem 3.4.5 *Let E be an entity, A be an architecture and $COMP$ be the set of diagnosis components computed from E and A. SIG is the set of signals defined in E and A. Then the following propositions hold:*

1. *The number of rules of the system description SD for $COMP$ and SIG is of order $o(|SIG| \cdot |COMP|^2)$.*
2. *The number of literals of the system description SD for $COMP$ and SIG is of order $o(|SIG|^2 \cdot |COMP|^2)$.*
3. *The time complexity of the system description is of order $o(|SIG|^2 \cdot |COMP|^2)$.*

Proof: Let $|COMP|$ be the number of components (concurrent statements) of Π and $|SIG|$ the number of signals defined in Π.

Since in the worst case all signals are used as input and output for every component the number of rules is of order $|COMP| \cdot |SIG|$ for every time point. Because of lemma 3.4.4 we have to consider $|COMP| + 1$ time points, the number of rules is of order $|COMP|^2 \cdot |SIG|$. With similar arguments we prove the second proposition.

In the worst case all signals are used as input and output for every component, leading to $|COMP| \cdot (|SIG| + 1) \cdot |SIG|$ literals for every time point. Since there are $|COMP| + 1$ points on the time axis, there are $o(|COMP|^2 \cdot |SIG|^2)$ literals. The third proposition follows directly from the second one by using the following arguments. The time complexity of resolution algorithms for propositional horn clauses is of linear order related to the number of literals. Therefore deciding if a contradiction can be derived using the FSD is limited by the squared number of components multiplied by the squared number of signals. ■

For programs with hundreds of components, the nonlinear size of the system description implies very long runtimes for the consistency checks required during the search for diagnoses. Since usually multiple consistency checks are required, a straightforward implementation of this system description results in completely unsatisfactory performance. Therefore, for implementation purposes this representation is transformed into a system description that employs a simplified approach for subsystems which are not involved in the kind of cyclic dependencies that led us to introducing distinct time values in the first case. The modified approach produces system descriptions that grow linearly in terms of system size. The transformation is presented in the following section (see also [SW95]).

3.5 Reducing Time Complexity

In this section we introduce a graph theoretic approach of reducing the system description size, i.e., the number of literals. First, a graph related to the underlying VHDL progam's functional dependencies is constructed. Properties of this graph lead to a new improved system description using strongly connected components. A subsection related to standard measurement selection approaches [dKW87] and restrictions following from that conclude the section.

3.5.1 Functional Dependencies as Graphs

This section describes the transformation of VHDL program components to a directed graph. We also give basic definitions about modeling VHDL programs and graph theory.

Similar to the previously introduced system description functional dependencies are used to present the way output ports of a component are connected with its input ports. The behavior of a component is given by by the value of the input and the internal state of the component. It can be defined using functions for every functional dependency. Note that not necessarily every input port of a functional dependency is used every time in computing the output. In some cases, a subset of the inputs may determine the calculated output value.

Definition 3.5.1 (Ports) *Let $COMP$ be the components of a VHDL program and SIG the set of signals of the program. The set of all ports of a component $C \in COMP$ is defined by $Ports(C) = \{(C, out) | out \in SIG \wedge \exists (out, in) \in FD(C)\}$. We call each pair (C, S) from $Ports(C)$ a* port *of C*

Note: We do not refer to (C, S) as an output port (as opposed to input port) of C, since in the following we never represent input ports directly. We therefore use the term *port* to refer to output ports exclusively.

The functions *In_Signals* and *Out_Signals* map the components in a FD Graph $\mathcal{G} = (V, E)$ to their sets of input and output signals, respectively, and are defined as

$$\forall C \in COMP : In\_Signals(C) = \bigcup_{(out,in) \in FD(C)} in$$

$$\forall C \in COMP : Out\_Signals(C) = \{S | (C, S) \in Ports(C)\}$$

Definition 3.5.2 (Component behavior) *Let C be a component and FD its set of functional dependencies. The behavior defined for C is a function defined for every element of FD which explains the output value as function of the associated input values with respect of the internal state.*

To reduce computational complexity, we introduce abstractions on the values propagated along connections, with the following being the most extreme case.

Definition 3.5.3 (Value-Abstract model) *A value-abstract model of the behavior of a component C is a model for C which uses only the values ok, $\neg ok$ or u (for* undefined*) to describe the state of the input and output signals of C.*

Given a value-abstract model, we thus have a straightforward notion of observations (identical to the definition in [FSW95]).

Definition 3.5.4 (Observation) *Let $TRACED$ be a set of traced signals, and $DISCR$ a set of discrepancies (obviously, $DISCR \subseteq TRACED$). The set of observations OBS is defined as*

$$OBS = \{\neg ok(s) | s \in DISCR\} \cup \{ok(s) | s \in TRACED \setminus DISCR\}$$

In addition, we require some basic graph theoretic definitions.

Definition 3.5.5 (Directed graph) *A directed graph $\mathcal{G}$ is a tuple (V, E) where V is the set of vertices and $E \subseteq 2^{V \times V}$ the set of directed edges. Given $(v_1, v_2) \in E$, we call v_1 the antecedens of v_2 and v_2 the successor of v_1.*

Note that this definition permits the existence of more than two arcs between two vertices. The number of arcs between two vertices is restricted by the number of elements of the label set L. We now define the mapping from a VHDL program to the graph expressing the dependencies in the program.

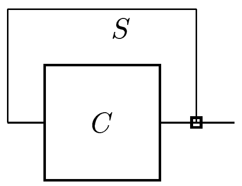
Definition 3.5.6 (FD Graph) *Let the definition of an VHDL program be given by the set of components, $COMP$, their functional dependencies FD, and the set SIG of connecting signals. Let $PORTS = \{Ports(C)|C \in COMP\}$. The FD Graph $\mathcal{G} = (V, E)$ for C with labeled edges is defined by*

1. $V = PORTS \cup SIG$
2. $\forall C \in COMP : \forall (out, in) \in FD(C) :$
   $((C, out), out) \in E \wedge \forall in_i \in in : (in_i, (C, out)) \in E$

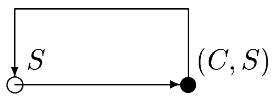
We also write $COMP(\mathcal{G})$ for the set of components $COMP$ that occur in $\mathcal{G}$. The vertices of the FD graph are either ports or signals. The edges are labeled with the signal name.

Example 5: This example shows the component view of a VHDL program and its FD Graph. The associated VHDL program could be, e.g., a clock generator `S <= not S after 10 ns` or something similar.

The graphical representation of the VHDL component with interconnections is given by:



The associated FD graph $\mathcal{G}$ is:



Note that only the label of the edge is represented in this graph. □

The following lemma states properties of FD graphs.

Lemma 3.5.1 (Properties of the FD graph) *Let $\mathcal{G} = (V, A)$ be the FD graph of a VHDL program. The graph $\mathcal{G}$ has the following properties:*

- $In\_Signals(C) = \{S|(S, (C, S')) \in E\}$
- $Out\_Signals(C) = \{S|(S', (C, S)) \in E\}$

- *The set of arcs contains only arcs from port nodes to signal nodes and from signal nodes to port nodes. There are no arcs connecting nodes of the same type.*
- *There exists not more than one arc from a particular port node to a particular signal node.*
- *The number of arcs is bounded by* $0 \leq |A| \leq |SIG| \cdot (1 + |PORTS|)$.
- *The number of port nodes is bounded by* $0 \leq |PORTS| \leq |SIG| \cdot |COMPS|$.

The proof of this lemma is trivial and omitted.

We first introduce how a functional system description related to the original one can be computed out of a FD graph.

3.5.2 Functional System Description

The Functional System Description is derived directly from the FD Graph.

Definition 3.5.7 (Functional System Description FSD) *Let* $\mathcal{G} = (V, E)$ *be the FD Graph of a VHDL program* Π. *The* Functional SD *(FSD) of the components and their connections is given by*

$\forall (C, S) \in V$:

1. *In_Signals(C)* $= \{S_1, \ldots, S_n\} \Rightarrow [ok(C) \wedge ok(S_1)_T \wedge \ldots \wedge ok(S_n)_T \rightarrow ok((C, S))_{T+1}] \in FSD$
2. $In\_Signals(C) = \emptyset \Rightarrow [ok(C) \rightarrow ok((C, S))_T] \in FSD$
3. $[ok(C) \rightarrow ok(S)_0] \in FSD$
4. $[\bigwedge_{((C_i,S),S) \in E} ok((C_1, S))_T \rightarrow ok(S)_T] \in FSD$

where $T \in \{0, \ldots, |COMP|\}$.

Traced signals and discrepancies are described in SD as shown in section 3.4 (see also [FSW95]). All theorems of that section hold for Functional SDs as well. Therefore the time T in a Functional SD must be a value between 0 and $|COMP|$. Note that except for notational differences in case 4 (we have omitted the use of the function $Drivers$ for defining this system description), the Functional system description FSD and the original system description SD ([FSW95]) are identical.

Theorem 3.5.2 *Let* P *be a VHDL program with components* $COMPS$, $\mathcal{G}$ *its FD Graph, and* OBS *a set of observations. A set* $\Delta \subseteq COMPS$ *is a diagnosis using the description SD iff it is a diagnosis using the the Functional System description* FSD.

Proof: (Sketch) The theorem holds if each diagnosis derived using SD can also be derived using FSD and vice versa. This can be shown using the property that for every rule of SD on P there is an equivalent rule in FSD on $\mathcal{G}$, and vice versa. ■

To achieve the goal of reducing computational complexity, we want to reduce the FD Graph to an acyclic one. This can be done by identifying cyclic subgraphs, reducing them to a single vertex (called a super-vertex") and computing the new edges. Each cycle in the graph must be covered by such a subgraph. Reducing a graph to a supergraph by eliminating all cycles individually is not feasible in general because of the possible exponential number of cycles. Accordingly, we must search for a supergraph that is acyclic, where each vertex can be computed in linear time, and which is suitable for diagnosis. The supergraph presented below is based on reduction of strongly connected components and is adaptable for diagnosis.

First, we need some standard auxiliary definitions. A sequence of arcs that connects two vertices of a graph is called a path.

Definition 3.5.8 (Path) *Let $\mathcal{G} = (V, E)$ be a directed graph and $n_1, n_2 \in V$ two vertices. A path P from n_1 to n_2 is a subset of E if $\exists (n_1, n) \in P$ such that $P \setminus \{(n_1, n)\}$ is a path from n to n_2.*

A path which is closed is called a (directed) cycle.

Definition 3.5.9 (Cycle) *Let $\mathcal{G} = (V, E)$ be a directed graph and $P = (n_1, n_2)$ a path. P is a cycle iff there are vertices $n, n_1, n_2 \in V$ such that $(n_1, n) \in P \wedge (n, n_2) \in P$.*

Next, we want to identify all pairs of components which are in a cycle of the graph together.

Definition 3.5.10 (Connectivity relation) *Let $\mathcal{G} = (V, E)$ be a directed graph. Two vertices v_1, v_2 are* κ-related *($v_1 R_\kappa v_2$) iff*

1. *$v_1 = v_2$, or*
2. *there is a cycle in $\mathcal{G}$ that contains both v_1 and v_2.*

Note that the relation R_κ is an equivalence relation that partitions $\mathcal{G}$ into subgraphs $\mathcal{G}_i^\kappa = (V_i, E_i)$.

Definition 3.5.11 (Strongly connected component) [3] *Let $\mathcal{G} = (V, E)$ be a graph and $\mathcal{G}' = (V', E')$ a subgraph of $\mathcal{G}$. $\mathcal{G}'$ is called a* strongly connected component *(SCC) of $\mathcal{G}$ iff it is equal to an element of the connectivity partitioning of $\mathcal{G}$ as defined by R_κ. If the connectivity partitioning contains only one component ($\mathcal{G}$), then we say that $\mathcal{G} = \mathcal{G}'$ is* strongly connected. *The* inputs *and* outputs *of $\mathcal{G}'$ are those outside vertices which are connected to vertices of $\mathcal{G}'$.*

[3] The term "component", as used in this context, stems from graph theory and does not in general correspond to any actual component of the VHDL program. If needed, we will make the distinction explicit in the text below.

$inputs(SCC) = \{v | (v, v') \in E \wedge v \notin V' \wedge v' \in V'\}$

$outputs(SCC) = \{v | (v', v) \in E \wedge v \notin V' \wedge v' \in V'\}$

Note that the inputs and outputs of a SCC can include both signal and port vertices.

A graph can be converted to a *strongly connected supergraph* by replacing every SCC of the given graph by a single vertex and remapping the edges accordingly.

Definition 3.5.12 (SCC supergraph) *Let $\mathcal{G} = (V, E)$ be a graph. The* SCC supergraph *(*SCCS*) $\mathcal{G}_s = (V_s, E_s)$ (called SCCS) associated with G is defined by*

1. $V_s = \{(V_i, E_i) | (V_i, E_i)$ *is a SCC of* $\mathcal{G}\}$
2. $E_s = \{((V_1, E_1), (V_2, E_2)) | (V_1, E_1), (V_2, E_2)$ *are SCC's of* $\mathcal{G} \wedge \exists (v_1, v_2) \in E : v_1 \in V_1 \wedge v_2 \in V_2\}$

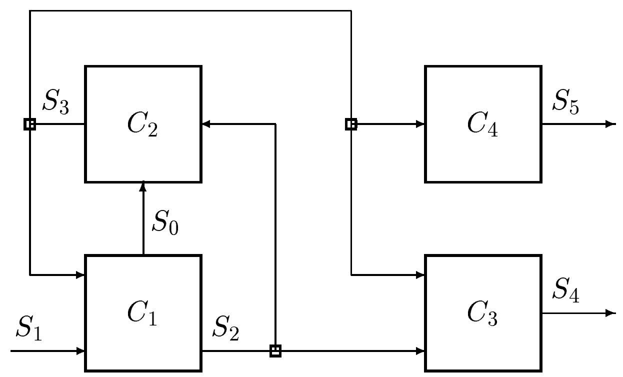
We write $Multiple(\mathcal{G}_s)$ to denote the set of all vertices in $\mathcal{G}$ which do not correspond to a single node in $\mathcal{G}$.

Lemma 3.5.3 (Properties of the SCCS) *Let $\mathcal{G}_s$ be the SCC supergraph of a graph $\mathcal{G}$, then the following propositions hold:*

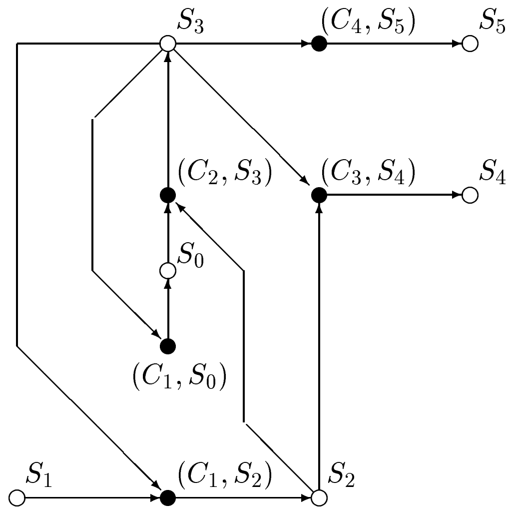
1. *The number of vertices of $\mathcal{G}_s$ is bounded by the number of vertices of $\mathcal{G}$ ($|V| \leq |V_s|$).*
2. *$\mathcal{G}_s$ contains no cycles.*

The following example shows the conversion from a VHDL program to the FD graph and then to the SCCS.

Example 6: This is a schematic depiction of a VHDL program containing a loop.



The FD Graph $\mathcal{G}\infty$ can be computed. (Vertices representing signals are shown in white.)



The graph $\mathcal{G}\infty$ includes 4 SCCs

$SCC_1 = [\{(S_1)\}, \emptyset]$

$$SCC_2 = [\{(C_1, S_2), (C_1, S_0), (C_2, S_3), S_0, S_2, S_3\},\\ \{((C_1, S_2), S_2), ((C_1, S_0), S_0),\\ ((C_2, S_3), S_3), (S_3, (C_1, S_2)),\\ (S_3, (C_1, S_0)), (S_0, (C_2, S_3)), (S_2, (C_2, S_3))\}]$$

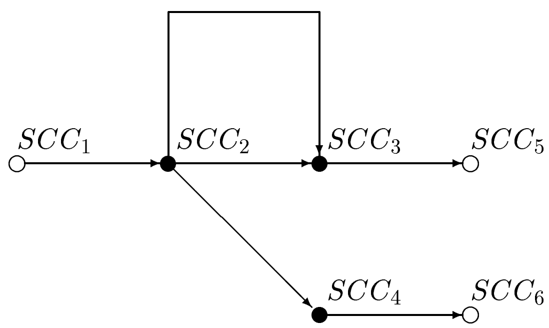
$SCC_3 = [\{(C_3, S_4)\}, \emptyset]$

$SCC_4 = [\{(C_4, S_5)\}, \emptyset]$

$SCC_5 = [\{S_4\}, \emptyset]$

$SCC_6 = [\{S_5\}, \emptyset]$

We now eliminate the cycle via the SCC supergraph transformation, resulting in the SCCS $\mathcal{G}\infty_S$. A vertex of $\mathcal{G}\infty_S$ is colored white if it contains no port vertices of the original graph (note that this implies it contains a single signal vertex).



The value of the function *Multiple* when applied to this example is $Multiple(\mathcal{G}\infty_s) = \{SCC_2\}$.

□

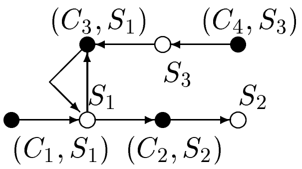
The second example is somewhat smaller than the first. We will use it later to show the way the system description is derived from the FD graph.

Example 7:

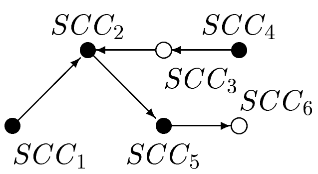
```
entity TESTER is
end TESTER;

architecture N6 of TESTER is
     signal S1, S2, S3:  std_logic := '0';
begin
     S1 <= '1' after 5 ns; – C1
     S2 <= not S1; – C2
     S1 <= S3 and S1 after 3 ns; – C3
     S3 <= '0', '1' after 2ns , '0' after 4 ns; – C4
end N6;
```

Computing the FD graph results in the directed graph:



The SCCS $\mathcal{G}_s = (V_s, E_s)$ derived from this graph looks as follows:



$SCC_1 = (\{(C_1, S_1)\}, \emptyset)$
$SCC_2 = (\{(C_3, S_1), S_1\}, \{((C_3, S_1), S_1), (S_1, (C_3, S_1))\})$
$SCC_3 = (\{S_3\}, \emptyset)$
$SCC_4 = (\{(C_4, S_3)\}, \emptyset)$
$SCC_5 = (\{(C_2, S_2)\}, \emptyset)$
$SCC_6 = (\{S_2\}, \emptyset)$

Vertices: $V_s = \{SCC_1, SCC_2, SCC_3, SCC_4, SCC_5, SCC_6\}$
Edges: $E_s = \{(SCC_1, SCC_2), (SCC_3, SCC_2), (SCC_4, SCC_3), (SCC_2, SCC_5), (SCC_5, SCC_6)\}$

The following table contains intermediate information helpful in computing the system description.

Component	In_Signals	Out_Signals
SCC_1	$\emptyset$	S_1
SCC_2	S_1, S_2	S_1
SCC_3	S_3	S_3
SCC_4	$\emptyset$	S_3
SCC_5	S_1	S_2
SCC_6	S_2	$\emptyset$

□

The following definition will be used later for classifying diagnosis problems with regard to measurement selection.

Definition 3.5.13 (Fully connected) *Let $\mathcal{G} = (V, A)$ be an FD graph for a VHDL program $\Pi = (Ent, Arch)$. We say that $\mathcal{G}$ is fully connected iff the following propositions hold:*

1. $\forall S \in SIG : \exists C \in COMP : (C, S) \in A$
2. $\forall S \in SIG : \forall (C, S') \in PORTS : ((C, S'), S') \in A \wedge (S, (C, S')) \in A.$

Lemma 3.5.4 *Let $\mathcal{G} = (V, A)$ be an FD graph. If $\mathcal{G}$ is fully connected then $\mathcal{G}$ is strongly connected.*

> **Proof:** Suppose $\mathcal{G}$ is fully connected. To show the correctness of the lemma we must only prove that every node of a fully connected graph can be reached by all other nodes. Let S be a signal node and N be a node. If N is a port node, then N must be directly connected to S because of the definition of fully connected. If N is a signal node, it must be connected to a port node. Because of the definition of full connection this port node is connected with all signal nodes, in particular also with S. ■

Finally, we show a result on the complexity of the functional system description FSD.

Theorem 3.5.5 *Let $\mathcal{G} = (V, A)$ be an FD graph for a VHDL program $\Pi = (Ent, Arch)$ and FSD its functional system description then $literals(FSD) = o(|COMP|^2 \cdot |SIG|^2)$.*

> **Proof:** Let $|COMP|$ be the number of components (concurrent statements) of Π and $|SIG|$ the number of signals defined in Π. In the worst case all signals are used as input and output for every component, leading to $|COMP| \cdot (|SIG| + 1) \cdot |SIG|$ literals for every time point. Since there are $|COMP| + 1$ points on the time axis, there are $o(|COMP|^2 \cdot |SIG|^2)$ literals. ∎

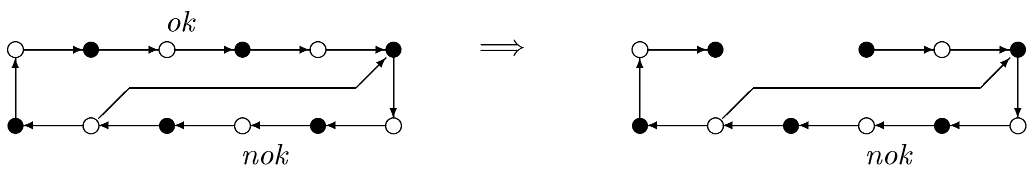
The time complexity of resolution algorithms for propositional horn clauses is of linear order related to the number of literals. Therefore deciding if a contradiction can be derived using the FSD is limited by the squared number of components multiplied by the squared number of signals.

3.5.3 Properties of the FD graph

Unfortunately, the size of a FSD is of order n^4 where n is the number of nodes in the graph. We now examine some properties which enable us to transform the FSD to an equivalent system description with linear complexity. We start with the observation that, since no actual signal values are observed, there is no need to keep track in the system description of signals that are actually ok.

Definition 3.5.14 *Let $\mathcal{G} = (V, A)$ be a FD graph as defined above, $TRACED$ the set of traced signals, and $DISCR$ the set of discrepancies. The* negative observations FD graph *$\mathcal{G}_{NO}$ is the subgraph of $\mathcal{G}$ induced by removing all signals occurring in $TRACED \setminus DISCR$ from V.*

In other words, all signals that are observed to be correct and all incident arcs are removed from the graph. The following example shows the effect of the transformation with two observations given. Note that in general, the negative observations FD graph does not need to be connected even if the original FD graph was.



Theorem 3.5.6 *Given $\mathcal{G}$, $TRACED$, $DISCR$, and $\mathcal{G}_{NO}$ be defined as above. Then $FSD(\mathcal{G}, TRACED, DISCR) = FSD(\mathcal{G}_{NO}, DISCR, DISCR)$.*

Proof: Removing signal nodes and arcs changes rule 1 of FSD, because removal of signals from the graph reduces the number of antecedent nodes for ports and therefore is equal to applying modus ponens to rule 1. The rule therefore remains correct. Rule 1 is the only rule where signals as input propagate their value to component outputs, so no other rules are affected. ■

With the possible reduction of signals in graph, we have to redefine the set of observations. From here on, the set of observations under FSD is defined as $OBS_{FSD} = \{\neg ok(s)_n | s \in DISCR\}$. In other words, all positive observations are omitted as they are no longer applicable to any signals in the graph. We continue to write simply OBS for the set of negative observations if the use of $\mathcal{G}_{NO}$ is understood.

3.5.4 A System Description with Linear Complexity

The SCCS_SD uses only information stored in the SCC supergraph. In particular, SCCS_SD can only be used if the observation set contains only *nok* facts, and in addition contains *nok* facts for all signals of a SCC. Note that the transformation to the negative observations graph guarantees exactly this condition. The figure below shows the transformations.

$$
\begin{tikzcd}
\Pi \arrow[r] & FDSet \arrow[r] & \mathcal{G} \arrow[r] \arrow[d] & FSD \\
OBS \arrow[rr] & & \mathcal{G}_T \arrow[r] & SCCS\_SD \\
 & & OBS_T &
\end{tikzcd}
$$

The original VHDL program Π is converted to its functional dependency set $FDSet$ and then to the associated FD graph $\mathcal{G}$. This graph can be used to directly derive the FSD system description. Instead, we convert $\mathcal{G}$ to $\mathcal{G}_{NO}$ depending on OBS according to Definition 3.5.14. The SCCS of $\mathcal{G}_{NO}$ is used as basis for computing the improved linear system description SCCS_SD.

3.5.5 System Description SCCS_SD

First, we need to extend the signal connection functions to SCC supergraphs.

Definition 3.5.15 (SCC Signal Connections) *Let $\mathcal{G}_s = (V_s, E_s)$ be an SCCS, and σ be an SCC of the original FD graph $\mathcal{G}$ corresponding to the subgraph (V_σ, E_σ) . We define* Ports *in the same manner as for the original graph:*

$\forall \sigma \in V_s Ports(\sigma) = \{(C, S) | (C, S) \in V_\sigma\}$

Using the results of section 3.5.3 we can derive the following improved system description.

Definition 3.5.16 (System description SCCS_SD) *Let $\mathcal{G}_{NO} = (V, E)$ be the negative observations FD Graph of a VHDL program, $\mathcal{G}_s = (V_s, E_s)$ the SCCS of $\mathcal{G}_{NO}$. For every node $\sigma \in V_s$, let (V_σ, E_σ) be the subgraph corresponding to σ in $\mathcal{G}_{NO}$. The* SCC System Description (SCCS_SD) *is given by*

1. *The outputs of a SCC are correct if the SCC and all inputs behave correctly.*

$$\forall v \in V_\sigma : [ok(\sigma) \wedge \bigwedge_{I \in inputs(\sigma)} ok(I) \rightarrow ok(v)] \in SCCS\_SD \tag{3.1}$$

2. *A SCC behaves correctly if all system components contained in it behave correctly.*

$$[\bigwedge_{(C,S) \in Ports(\sigma)} ok(C) \rightarrow ok(\sigma)] \in SCCS\_SD \tag{3.2}$$

3. *A signal cannot be correct and incorrect at the same time.*

$$ok(S) \wedge nok(S) \rightarrow \perp \tag{3.3}$$

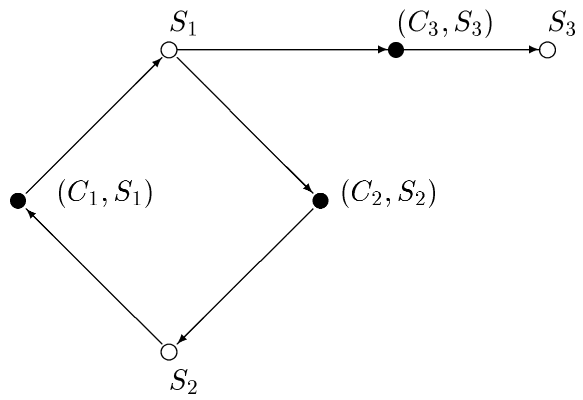
Note: In the case of a SCC that is a singleton set containing a signal (such as SCC_5 and SCC_6 in the example), some of the sentences above are trivially true. For example, according to rule 3, such a SCC always behaves correctly (since the antecedent is empty).

Under the new SD, the derivation of observations from traces is straightforward. Note that since SCCS_SD is derived from the negative observations graph, no positive observations are possible.

Definition 3.5.17 (Observations under SCCS_SD) *For a given set $TRACED$ and $DISCR$, the set OBS according to $SCCS\_SD$ (also written as OBS_{SCCS}) is given by*

$OBS = \{nok(S) | S \in DISCR\}$

Example 8: To show how the new improved system description works consider the following example graph.



The three strongly connected components are $SCC_1 = (\{S_1, S_2, (C_1, S_1), (C_2, S_2)\}, \ldots)$, $SCC_2 = (\{(C_3, S_3)\}, \emptyset)$, and $SCC_3 = (\{S_3\}, \emptyset)$.

Depending on the traced signals and discrepancies, different diagnoses will be derived with the FSD.

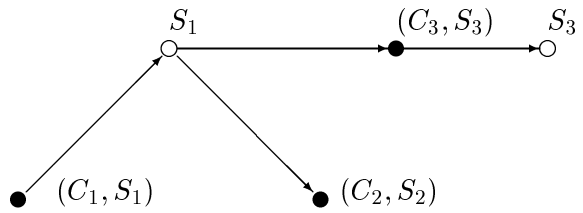
1. $TRACED = DISCR = \{S_3\}$ From the definition of the FSD, it follows that all three components C_1, C_2, C_3 are single diagnoses.
2. $TRACED = \{S_1, S_3\}, DISCR = \{S_3\}$ The only possibility for a single diagnosis is C_3.
3. $TRACED = \{S_2, S_3\}, DISCR = \{S_3\}$ In this case there are two single diagnoses, C_1 and C_3.

In all cases the new SCCS_SD will answer the same results as the FSD iff the original graph is converted to the associated negative observation graph before applying the system description rules.

Now consider case 3, and assume that the SCCS_SD is derived from the original graph. Then all three components are returned asn (incorrect) single diagnoses. We take a closer look at case 3.

From the first rule, it follows that the output (C_3, S_3) can only be correct iff SCC_1 behaves correctly. This on the other hand happens iff C_1 and C_2 are assumed to be correct. Therefore C_1 and C_2 are two single explanations for example case 3. The third minimal diagnosis follows directly from the system description and the observations by assuming that C_1 and C_2 must behave correctly.

Let us now remove all correct signals from the graph before deriving the system description. In this case the graph has the following appearance:



The resulting graph has no SCC with more than one vertex. It can be easily seen that in this case the correct result can be derived using the SCCS_SD.

□

3.5.6 Equivalence of SCCS_SD and FSD

The next goal is to prove that the SCCS_SD system description produces the same results as FSD.

We call a set Δ a diagnosis candidate if Δ is of the form $\{ok(c_1), \ldots, ok(c_n)\}$ such that $c_i \in COMP$, for $i = \in \{1, \ldots, n\}$. (We do not specify which SD we mean if it is clear from context.)

Definition 3.5.18 (Derivation Graph) *Let $\mathcal{G}$ be a FD graph, $SCCS\_SD$ its system description, with $\mathcal{G}_s = (V_s, E_s)$ the resulting negative observations supergraph, and OBS the resulting set of negative observations, and let s be a signal from $\mathcal{G}$. The* derivation graph $D = (VD, ED)$ *of $ok(s)$ is defined as follows:*

1. *$ok(s) \in VD$*
2. *For every vertex v such that $v \in \mathcal{G}$ (v is from the original FD graph, not a composite SCC) and $ok(v) \in VD$, if σ is the SCC including v, then $ok(\sigma) \in VD$, and there is an arc $(ok(\sigma), ok(v))$ in ED. In addition, for every input I of σ, $ok(I) \in VD$, and there is an arc $(ok(I), ok(s')) \in ED$.*
3. *for every SCC $\sigma \in V_s$ and port (C, S) in σ, $ok(C) \in VD$, and there is an arc $(ok(C), ok(\sigma)) \in ED$.*

Note that is not possible that $ok(s) \in OBS$, since there are no positive observations in OBS. Also, note that D must be finite, since $\mathcal{G}_s$ contains no cycles. Furthermore, all source vertices of D are of the form $ok(C)$ or $ok(\sigma)$. A source vertex of the form $ok(\sigma)$ only occurs if σ is a singleton SCC containing a single signal. Remember that in the latter case, $ok(\sigma)$ holds automatically (see note to Definition 3.5.16). Finally, every component C for which a path exists in $\mathcal{G}_s$ from a port (C, S') to S is a source vertex of D.

Lemma 3.5.7 *Let D be the derivation graph for $ok(s)$, Δ a diagnosis candidate. $SCCS\_SD \cup OBS \models ok(S)$ iff $ok(C)$ holds for all C such that $ok(C)$ is a source vertex of D.*

> **Proof:** The graph corresponds directly to a sequence of applications of rules 1 and 2 from $SCCS\_SD$. For the if direction, all antecedents are true. The facts of form $ok(C)$ are true by assumption, and the facts of form $ok(\sigma)$ are automatically true (see the note to Definition 3.5.16). Therefore the derivation described by the graph is valid.
>
> The only if direction follows immediately from the observation that if $ok(s)$ is derivable, the derivation graph describes the only way in which it can be derived using the horn clause rules 1, 2, and 3. Only $ok(C)$ facts can be used to start the derivation, since there are no positive observations in $\mathcal{G}_s$. ■

Lemma 3.5.8 *$\mathcal{G}_{NO} = (V, E)$ be a negative observations graph, and D be the derivation graph for $ok(s)$. For every component c such that there is a path from from $ok(c)$ to $ok(s)$ in D, there is a path from some port (c, s') ($s' \in V$)to s in $\mathcal{G}_{NO}$.*

> **Proof:** We designate the SCC of s by σ_s, and the SCC of (c, s') by σ_c. Note that in FD_{NO} there exists a path between any two vertices in a SCC σ since σ is strongly connected.

First, assume that $\sigma_c = \sigma_s$. Then, there exists a path between (c, s') and s since they are in the same SCC. Second, if $\sigma_c \neq \sigma_s$ (i.e., there is no port of c in σ_s), there must exist a sequence of SCC's in D of the form $< \sigma_c = \sigma_1, \sigma_2, \ldots, \sigma_n = \sigma_s >$. Note that every arc in D that connects vertices corresponding to different SCC's corresponds to at least one arc in FD_{NO}, since these arcs are defined in terms of inputs to SCC's (rule 1 of Definition 3.5.16). Therefore, for every SCC σ_i in the sequence except σ_1, there is an arc from some vertex in σ_{i-1} to some vertex v_{i1} in σ_i, and for every σ_i except σ_n there is an arc from some vertex v_{i2} to a vertex in σ_{i+1}, and because σ_i is an SCC, there is a path from v_{i1} to v_{i2}, as well as a path from (c, s') to v_{12} in σ_1 and a path from v_{n1} to s in σ_n. Therefore, there is a path in FD_{NO} from (c, s') to s. ■

Lemma 3.5.9 *Let $COMP$, SIG, and FD be the description of a VHDL program, $\mathcal{G} = (V, A)$ its FD graph, $TRACED$ a set of traced signals and $DISCR$ a set of detected discrepancies, with $\mathcal{G}_s$ being the resulting negative observations SCCS, and OBS the resulting set of observations. Given a diagnosis candidate Δ, if $\Delta \cup FSD \cup OBS_{FSD}$ is consistent, then $\Delta \cup SCCS\_SD \cup OBS_{SCCS}$ is consistent.*

Proof: Assume that $C(\Delta) \cup SCCS\_SD \cup OBS_{SCCS}$ is inconsistent. In that case, it follows that there is some $S \in SIG$ such that $\neg ok(S) \in OBS$, but $ok(S)$ can be derived. Let $C = \{C_1, \ldots, C_m\}$ be the set of all components C_i such that that a path exists in $\mathcal{G}_s$ from a port (C_i, S') to S, for some $S' \in SIG$. Now, since $ok(S)$ can be derived, we know from Lemma 3.5.7 that, $ok(C_i)$ holds for every $C_i \in C$. From the definition of OBS_{FSD}, we know that $\neg ok(S)_n$ holds, and from Lemma 3.4.4, we know that $ok(S)_n$ can be derived since $ok(C_i)$ holds for every $C_i \in C$. Therefore, $C(\Delta) \cup FSD \cup OBS_{FSD}$ would be inconsistent as well, which contradicts the initial assumption. ■

Lemma 3.5.10 *Let $COMP$, SIG, and FD be the description of a VHDL program, $\mathcal{G} = (V, A)$ its FD graph, $TRACED$ a set of traced signals and $DISCR$ a set of detected discrepancies, with $\mathcal{G}_s$ being the resulting negative observations SCCS graph, and OBS the resulting set of negative observations. Given a diagnosis candidate Δ, if $\Delta \cup FSD \cup OBS_{FSD}$ is inconsistent, then $\Delta \cup SCCS\_SD \cup OBS_{SCCS}$ is inconsistent.*

Proof: If $C(\Delta) \cup FSD \cup OBS_{FSD}$ is inconsistent, we know that for some signal s such that $\neg ok(s)_n \in OBS_{FSD}$, $C(\Delta) \cup FSD \cup OBS_{FSD} \models ok(s)_n$. From Corollary 3.4.2, we know that all components c for which a path exists from a port (c, s') to s must be ok. Therefore, according to Lemma 3.5.8, all components which are source vertices in the derivation graph D for $ok(s)$ using $SCCS\_SD$ are ok as well. This implies $C(\Delta) \cup SCCS\_SD \cup OBS_{SCCS} \models ok(s)$, which produces the desired inconsistency since $\neg ok(s) \in OBS_{SCCS}$. ■

Theorem 3.5.11 *Let $COMP$, SIG, and FD be the description of a VHDL program, $\mathcal{G} = (V, A)$ its FD graph, $TRACED$ a set of traced signals and $DISCR$ a set of detected discrepancies. For every $\Delta \subseteq COMP$ it holds that Δ is a minimal diagnosis under $FSD(\mathcal{G}, TRACED, DISCR)$ iff it is a minimal diagnosis under $SCCS\_SD(\mathcal{G}_{TO}, DISCR, DISCR)$*

Proof: We first prove that Δ is indeed a diagnosis for both system descriptions.

From Lemma 3.5.9, we know that Δ is a diagnosis for SCCS_SD if it is a diagnosis for FSD. For the only if direction, assume that Δ is a diagnosis for SCCS_SD but not for FSD. That means that $C(\Delta) \cup FSD \cup OBS_{FSD}$ is inconsistent. But then, according to Lemma 3.5.10, $C(\Delta) \cup SCCS\_SD \cup OBS_{SCCS}$ is inconsistent as well, which contradicts the assumption that Δ is a diagnosis for SCCS_SD.

Since the set of diagnoses for both system descriptions is the same, it follows that the set of minimal diagnoses is equal as well. We show only one direction. Assume Δ is a minimal diagnosis for SCCS_SD but not for FSD. That implies there exists a $\Delta' \subset \Delta$ that is a diagnosis for FSD. From Lemma 3.5.9 it follows that Δ' is also a diagnosis for SCCS_SD, and therefore Δ is not minimal for SCCS_SD, which contradicts the assumption. ■

Finally we show that the new system description indeed results in better time complexity than the old one.

Theorem 3.5.12 (Time complexity of SCCS_SD) *Let $COMP$, SIG, and FD be the description of a VHDL program, $\mathcal{G} = (V, A)$ its FD graph, $TRACED$ a set of traced signals and $DISCR$ a set of detected discrepancies. The time complexity of the associated system description $SCCS\_SD$ is of order $|COMP|$ multiplied by $|SIG|$.*

Proof: Let c be the number of components ($c = |COMP|$) and s be the number of signals ($s = |SIG|$). To prove the theorem we investigate the number of literals used for the SCCS_SD in the worst case.

First consider rule 2. The number of SCC n_{scc} is limited by the s and c and must be greater or equal 1. So $1 \leq n_{scc} \leq c + s$. We further know that this number can be computed by adding the number of all components used within a SCC. Assume that the number of components for the i-th SCC is given by c_i. Then the number of literals used in rule 2 l_2 is given by $l_2 = \sum_{i=1}^{n_{scc}} (c_i + 1)$. From this follows $l_2 = \sum_{i=1}^{n_{scc}} c_i + \sum_{i=1}^{n_{scc}} 1$, $l_2 = c + n_{scc}$, and finally $l_2 \leq 2 \cdot c + s$.

Secondly consider rule 3. The number of literals for this rule l_3 is limited by the number of signals, such that $l_3 = 2 \cdot s$.

To conclude the proof we must compute the number of literals for rule 1 (l_1). This number is given by the number of inputs and the number of vertices used in a SCC, such that $l_1 = \sum_{i=1}^{n_{scc}} (1 + input_i + vertices_i)$. We do not exactly compute this sum but give an upper bound. Since an input of one SCC can only be an vertex of another SCC the sum of inputs and vertices must be limited by the maximum number of vertices of $\mathcal{G}$. This number is given by $s \cdot c$. So there must be a constant k such that $l_1 \leq k \cdot s \cdot c$.

Because the number of literals is of order $s \cdot c$ and the time complexity depends linear on the number of literals, the theorem must hold. ■

The new system description has a linear time complexity depending on the number of components.

3.6 Measurement selection using value-abstract system descriptions

Measurement selection is used for discriminating between diagnosis candidates. DeKleer [dKW87] introduced a measurement selection algorithm that computes the most probable measurement selection point. The computational complexity of the original algorithm is not suitable for practical usage. The simple measurement selection model uses only the information about leading diagnoses and does not change component probabilities after introducing new observations[4].

We know the following fact:

$\forall S \in Signals : C(\Delta) \cup SD \cup OBS \models value(S, v)$ for $v \in domain(S)$

The possibility of measuring a particular value for a signal depends on the correctness of the diagnosis assumption. Assuming that a diagnosis is correct leads to the following probability:

$$p(value(S, v)|\Delta) = \begin{cases} 0 & C(\Delta, COMP) \cup SD \cup OBS \not\models value(S, v) \\ 1 & otherwise \end{cases}$$

The overall possibility of measuring a value given a leading diagnosis accumulates to:

$$p(value(S, v)) = \sum_{\Delta} p(value(S, v)|\Delta) \cdot p(\Delta)$$

Various possible definitions of the probability of a diagnosis can be given. We assume the existence of a fault probability for every component.

$$p(\Delta) = \sum_{C \in COMP - \Delta} (1 - p(C)) + \sum_{C \in \Delta} p(C)$$

where $p(C)$ is the fault probability of the component C.

To choose the right measurement selection point, we use the entropy function. The idea behind this is that we want measure only signals where the measured value discriminates between as many (leading) diagnoses as possible.

$$E(S) = - \sum_{v \in domain(S)} p(value(S, v)) \cdot \log p(value(S, v))$$

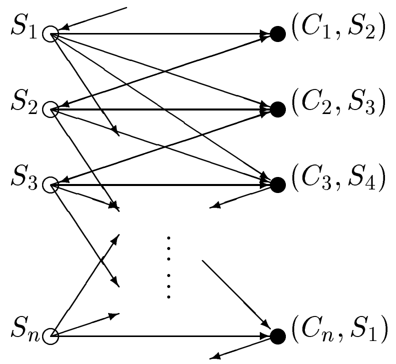
We prefer a signal S over a signal S' iff $E(S) > E(S')$.

[4] Introducing new observations restricts the diagnosis space. Components with high fault probability are possibly not longer element of a leading diagnosis.

3.7 Limits of the functional system description FSD

The Functional System Description introduced in Section 3.5.2 places severe restrictions on the utility of measurement selection, due to the very abstract domains and component behavior. In this section, we show several cases where the approach is ineffective.

Consider the following example graph.



This graph represents the worst case graph for the Functional SD in the sense that the system description does not differentiate between signals for measurement. In other words, if a VHDL program has the above FD graph, the measurement selection algorithm will return the set of all signals for checking. In such a case, no additional information can be given to the user (designer).

Theorem 3.7.1 *Let $\mathcal{G} = (V, A)$ an FD graph for a VHDL program $\Pi = (Ent, Arch)$. If $\mathcal{G}$ is fully connected and FSD is used, a measurement selection algorithm cannot compute different measurement selection results.*

> **Proof:** Let $\mathcal{G}$ be a fully connected graph. Then we know that $\mathcal{G}$ can be transformed into a SCC supergraph containing only one node. From the system description we know that in such a case all components can cause a discrepancy. Let us first consider the case in which we have detected exactly one discrepancy. For simplicity we assume that every concurrent statement is associated with exactly one port. Then $|COMP| = |SIG| = |PORTS| = n$ holds. If a signal S is incorrect, $\neg ok(S)_n \in SD$. For each $C_i \in COMP$, $\{C_i\}$ is a possible diagnosis.
>
> To prove that the measurement selection cannot compute relevant results, we show that every diagnosis computes the same values for every signal. Let $\{C_i\}$ be a diagnosis. Then the system description includes $ok(C_1), \ldots, ok(C_{i-1}), \neg ok(C_i), ok(C_{i+1}), \ldots, ok(C_n)$ as facts. Let (C_j, S_j) be the to C_i associated port. Then we can derive only $ok(S_j)_0$ for $j \neq i$. Because of the system description and because $\neg ok(C_i)$ is true we cannot derive anything more. Therefore

using all diagnoses leads to conclusion that every value of every diagnosis is equally often derivable ($|COMP| - 1$ times) and a measurement selection that uses no fault probability cannot differentiate between different signals.

With the same arguments we can prove that the theorem holds if multiple discrepancies have been detected. To finish the proof, we must show that the theorem holds also after the detection of correct signals. In this case the measurement selection cannot discriminate between signals that are known to be correct or incorrect. The proof is simple. Remember that a correct signal S introduces values $ok(S)_0, \ldots, ok(S)_{|COMP|}$. Because of the system description we can derive a new graph by removing node S, all arcs connected to S, all port nodes that have S as successor node and their arcs. This graph is again fully connected. By applying the rules of node removal to all correct signals we get a fully connected graph which uses only signal nodes that are incorrect or unknown. Now we can use the first part of the proof to show the correctness of the theorem. ■

3.8 Implementation and Result

VHDLDIAG parses VHDL source files to generate a set of diagnosis components and their interconnections. Diagnoses are computated with standard techniques described in [Rei87, GSW89, dK91]. The diagnosis generator uses the component description and a set of discrepancies to build a hitting set dag and then all minimal diagnoses.

To prune the hitting set dag we allow only the generation of the dag up to a fixed size or depth. Only diagnoses with size up to the dag depth are computed. To restrict the set of diagnoses to a relevant focus, we also allow the insertions of fault probabilities for each diagnosis component computed by a simple heuristic based on its complexity.

Additional measurements for discrimination are selected using the minimum entropy approach introduced in [dKW87]. In this context, making additional measurements means specifying additional signals to be traced in the next simulation cycle.

The presented diagnosis approach has been tested on a number of examples, including actual full-size ASIC designs. The VHDLDIAG system has beem in use in various design projects. Although developers are only just learning how to use the system effectively, we can report the following impacts on the design cycle: The total effort spent on the simulation-design-debug cycle has been reduced by 25%. Higher savings are expected as users become more familiar with the tool. Computation costs for simulation runs were reduced by 15%.

The following data points describe one of the ASICs used for testing the VHDLDIAG system (which is now being used in real world applications).

Components at top level	379
Number of interface signals at top level	210
Number of signals at top level	412
Maximum number of levels	5
Average number of levels	~ 2
Size of source code	6 MB
Number of gates in produced circuit	$> 100{,}000$

To show the capability of the introduced diagnosis approach, we developed two test situations. In the first case, we used two different version of the ASIC implementation, which are structurally equivalent, and compared them by applying test cases. Discrepancies are detected using a waveform compare tool. From them we are able to find the faults in the program. The following table shows the measured runtimes (on a SPARC 10/30) for locating a fault.

Simulation time	10 minutes
Compare time	20 minutes
Diagnosis time	2 minutes
Diagnosis cycle	32 minutes

Because of the knowledge of all signal values over time, VHDLDIAG computes exactly one minimal diagnosis per level. This diagnosis was always the expected one. A complete regression test lasts about one hour per test case, whereas manual examination of test results lasts several hours for only the top level of an ASIC. VHDLDIAG can diagnose faults at all levels of the ASIC design due to the ability to incorporate all traced signals in the solution (which is far beyond the ability of a human studying a trace).

In the second case, we used only one ASIC implementation and user input about signal correctness to find the faults. Every question VHDLDIAG presents to the user reduces the number of diagnoses but increases the overall diagnosis time, due to the need to recompute diagnoses. The effect of VHDLDIAG in this domain is not as good as it is in the regression test domain, but because of the ability of VHDLDIAG to focus the designer's attention on the relevant signals by using automatic measurement selection, debugging time can again be saved.

All other ASIC examples required at most 3 minutes to compute all minimal diagnoses at the top level. The number of components and signals decreases for lower levels in the ASIC design. At the lowest level we have measured diagnosis times below one second.

Diagnosis results strongly vary depending on the number of signals which could be classified as ok or $\neg ok$. If this number increases, the expected diagnosis can be computed with more accuracy.

VHDLDIAG has shown that a simple model used for model-based diagnosis can be good enough for utilization in the VHDL design process. Especially in the domain of regression tests, fault-finding time can be reduced and quality can be increased. Using VHDLDIAG while implementing a new program helps the designer by focusing on relevant signals and components. The diagnosis runtimes achieved by the VHDLDIAG prototype are short enough to permit its use in interactive diagnosis sessions even for very large VHDL programs.

Chapter 4

Software Debugging using Program Semantics

4. Der Gedanke ist der sinnvolle Satz.

Ludwig Wittgenstein, 1921
Tractatus logico-philosophicus
Logisch-philosophische Abhandlung

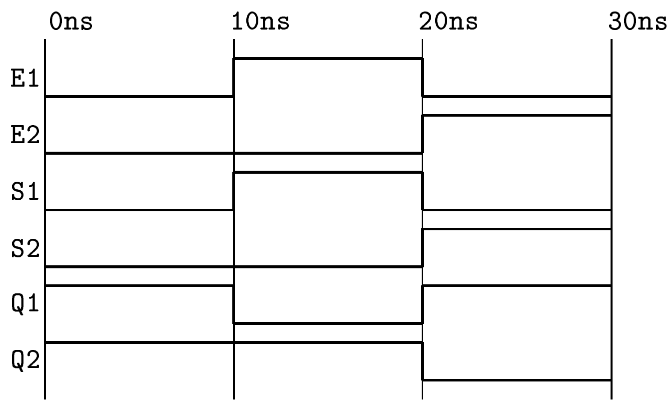
As stated in the previous chapter, the approach of using only functional dependencies for diagnosis sometimes delivers unsatisfactory results. We therefore extend the basic approach by introducing program semantics. This is done by using evaluation traces and converting them components and connections for the sequential part of VHLD. Used components are basic components (such as AND) with known behavior. Unfortunately evaluation traces cannot be used for diagnosis of the concurrent part of VHDL because processes may or may not be active during a simulation cycle. If a process is not active it does not appear in the associated evaluation trace. Consider now the case where a signal change its value because of backpropagation while computing diagnosis. In this case a process (formally not active) should be active. Such a process is not element of the trace (and of the associated diagnosis system). We examplify this problem using a short example.

Example 9: Consider the following VHDL program implementing a simple nand flip flop with an additional testbench.

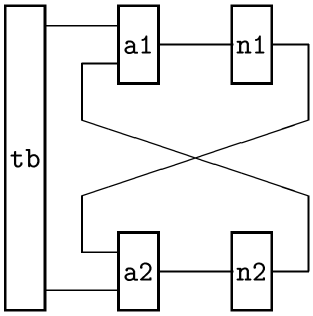
```
entity ff is
end ff;
architecture beh of ff is
     signal E1,E2,Q1,Q2,S1,S2: bit ;
begin
     a1: process
```

```
    begin
        S1 <= E1 and Q2;
        wait on E1, Q2;
    end a1;
    n1:  process
    begin
        Q1 <= not S1;
        wait on S1;
    end n1;
    a2:  process
    begin
        S2 <= E2 and Q1;
        wait on E2, Q1;
    end a2;
    n2:  process
    begin
        Q2 <= not S2;
        wait on S2;
    end a1;
    tb:  process
    begin
        E1 <= '0', '1' after 10ns, '0' after 20ns;
        E2 <= '0', '1' after 20ns;
        wait until false;
    end tb;
end beh;
```

The behavior of the signals defined in the architecture `beh` should be as follows:



The static structure of the example (processes and their connections via signals) can be expressed by:



The dynamic behavior while computing signal values is given by the VHDL semantics and can be expressed by the following table. Signal values and active processes resulting from signal changes are presented. Every row with active processes denotes a simulation cycle. The results of the cycle, i.e., the signal values, are given on the right side. A row without active process denotes the start of a simulation cycle where signal values are changed using the previously computed projected output waveforms of signals.

Time	Active Processes	E1	E2	Q1	Q2	S1	S2
0ns		'0'	'0'	'0'	'0'	'0'	'0'
	a1, a2, n1, n2, tb	'0'	'0'	'1'	'1'	'0'	'0'
	a1, a2	'0'	'0'	'1'	'1'	'0'	'0'
10ns		'1'	'0'	'1'	'1'	'0'	'0'
	a1	'1'	'0'	'1'	'1'	'1'	'0'
	n1	'1'	'0'	'0'	'1'	'1'	'0'
	a2	'1'	'0'	'0'	'1'	'1'	'0'
20ns		'0'	'1'	'0'	'1'	'1'	'0'
	a1, a2	'0'	'1'	'0'	'1'	'0'	'0'
	n1	'0'	'1'	'1'	'1'	'0'	'0'
	a2	'0'	'1'	'1'	'1'	'0'	'1'
	n2	'0'	'1'	'1'	'0'	'0'	'1'
	a1	'0'	'1'	'1'	'0'	'0'	'1'

As can be seen, in each simulation cycle of this example (excluding the first one) only a subset of all processes is active.

□

As stated above, diagnosis of concurrent statement should be done using traditional hardware diagnosis approaches (see [HD84, MNPH90] for example) because otherwise the diagnosis results need not to be correct. In [MNPH90] the application of VHDL for building system descriptions for diagnosis is discussed, and the description of a temporal constraint propagation system which can be also used for software diagnosis in this domain is given. See also [Wil86, VKvB86] for an introduction to temporal constraint propagation.

Because the concurrent part of **RVHDL** can be diagnosed using traditional MBD approaches we concentrate on software debugging of the sequential part of VHDL. In the remainder of this section we present the conversion of sequential statements to components and connections, show the application of this concept, and discuss improvements.

4.1 Converting Traces of Sequential Statements to Diagnosis Components

Sequential statements can be viewed as components where signals and variables represent the connection between them. In our case, sequential statements are not alone viewed as components. Instead, expressions and subexpressions are also converted to components. However, only sequential statement are used for diagnosis[1].

In the following we introduce a function *conv* defining how conversion should take place. To do this we assume the existence of a function $CEnv$ associating connections to signals and variables, a function $Comps$ answering the converted components, and a function SD returning the current system description. The system description stored in SD is called **SD_SEQ** standing for diagnosis of SEQuential statements.

Components are expressed by pairs (St, Env) where St is the related statement and Env the actual environment used during conversion. A rule expressing the component's behavior is added to the system description. The statement `S <= (I1 and I2) or I3;` is converted to the component $c =$ (`S <= (I1 and I2) or I3;`, Env) and the related rule $ok(c) \Rightarrow (out_c = or(and(in_1, in_2), in_3))$ expressing the behavior of c. The inputs in_1, in_2, in_3 are related to signals or variables used in the expression. Note that in_i is connected to a component's output.

We present the conversion function in a top-down manner. First we show how processes of the form

$P = Id$ `process` PD `begin` $St_1 \dots St_n$ `end process` Id

are converted into diagnosis systems.

$conv(P, Env) = conv(St_1; \dots St_n;, Env, CEnv, \{\}, SD) = (Env', CEnv', Comps', SD')$

with $CEnv$ and SD defined as follows:

$\forall S \in target(P) \cup vars(P) : CEnv(S) = out(S, Env) \wedge \{out(S, Env) = Env(S) \in SD$

4.1.1 Converting Statements

- *Sequential Statement Blocks* Blocks are converted into components by sequentially converting each statement into a component and using variables and signals as connections.

  $inputs(St_1; \dots St_n;) = \bigcup_{i=1}^{n} inputs(St_i)$

  $outputs(St_1; \dots St_n;) = \bigcup_{i=1}^{n} outputs(St_i)$

  $conv(St_1; \dots St_n;, Env, CEnv, Comps, SD) =$

[1]Expression components can also be used for diagnosis causing only small changes of the conversion function but increasing the number of diagnosis components and hence the overall computation time.

$conv(St_2;\ldots;St_n;,Env',CEnv',Comps',SD')$

with

$(Env',CEnv',Comps',SD') = conv(St_1,Env,CEnv,Comps,SD)$

- *Signal Assignments* Signal assignments are converted into a component called *ASSG_COMP*. The *ASSG_COMP* iff said to behave correct uses the internal function to compute a single output value for the given input values. Inputs are those signals and variables used in the expression. The output is a connection representing the target signal. The time expression is not taken into account because the focus of this thesis is in finding functional faults rather than timing faults. This is an important topic for future research.

  $inputs(T\texttt{<=}E\ \texttt{after}\ TE) = signals(E)$

  $outputs(T\texttt{<=}E\ \texttt{after}\ TE) = \{T\}$

  $conv(T\texttt{<=}E\ \texttt{after}\ TE,Env,CEnv,Comps,SD) =$
  $\quad (Env,CEnv',Comps',SD'')$

  where

  $c = (T\texttt{<=}E\ \texttt{after}\ TE,Env)$

  $Comps' = Comps \cup \{c\}$

  $SD'' = SD' \cup \{ok(c) \Rightarrow (out_c = Top)\}$

  $\forall S \in SIG \cup VAR \cup DRIVER : S \neq T \Rightarrow CEnv'(S) = CEnv(S)$ and $CEnv'(T) = out_c$

  with

  $(V,Top,SD') = conv_E(E,Env,CEnv,Comps,SD)$

- *Variable Assignments* Conversion of variable assignments is similar to the conversion of signal assignments, except that the environment for the variable must be changed, and that they are converted into a component *VASSG_COMP*. The functionality is the same as for *ASSG_COMP*.

  $inputs(T\texttt{:=}E) = signals(E)$

  $outputs(T\texttt{:=}E) = \{T\}$

  $eval(T\ \texttt{:=}\ E,Env,CEnv,Comps,SD) = (Env',CEnv',Comps',SD'')$

  where

  $c = (T\ \texttt{:=}\ E,Env)$

  $Comps' = Comps \cup \{c\}$

  $SD'' = SD' \cup \{ok(c) \Rightarrow (out_c = Top\}$

  $\forall S \in SIG \cup VAR \cup DRIVER : S \neq T \Rightarrow CEnv'(S) = CEnv(S)$ and $CEnv'(T) = out_c$

  with

  $(V,Top,SD') = conv_E(E,Env,CEnv,Comps,SD)$

  $Env' = Env[(T,V)]$

- *Conditionals* Conditionals change signals depending on the value of a condition. Every branch can change other signals. In general it is not possible to trace both branches because one branch may lead to an infinite loop. If this cannot happen, we can trace (and convert) both branches, possibly leading to less diagnoses. If infinite loops are possible, we convert only the branch which is element of the evaluation trace.

  $inputs(\texttt{if } E \texttt{ then } SB_1 \texttt{ else } SB_2 \texttt{ end if}) = signals(E) \cup inputs(SB_1) \cup inputs(SB_2)$

  $outputs(\texttt{if } E \texttt{ then } SB_1 \texttt{ else } SB_2 \texttt{ end if}) = outputs(SB_1) \cup outputs(SB_2)$

  - *Full conversion* A fully converted conditional statement can be used to restrict the search space by defining an additional fault mode related to a wrong conditional expression. We call this mode *wcond*.

    $conv(\texttt{if } E \texttt{ then } SB_1 \texttt{ else } SB_2 \texttt{ end if}, Env, CEnv, Comps, SD) =$
    $\quad (Env', CEnv', Comps', SD')$

    with

    $(Env_1, CEnv_1, Comps_1, SD_1) = conv(SB_1, Env, CEnv, Comps, SD)$
    $(Env_2, CEnv_2, Comps_2, SD_2) = conv(SB_1, Env, CEnv, Comps, SD)$
    $(V, SD'', Top) = conv_E(E, Env, CEnv, Comps, SD)$

    If $V = true$ then let $Env' = Env_1$ otherwise let $Env' = Env_2$.

    $c = \texttt{if } E \texttt{ then } SB_1 \texttt{ else } SB_2 \texttt{ end if}, Env)$
    $Comps' = Comps \cup Comps_1 \cup Comps_2 \cup \{c\}$
    Let $s_1, \ldots, s_n$ be element of the set of all outputs of the conditional statement.

    $$SD' = SD \cup SD'' \cup SD_1 \cup SD_2 \cup \\ \bigcup_{i=1}^{n}\{ \\ ok(c) \Rightarrow (in_E = true \rightarrow (out_i = in_{Ti})), \\ ok(c) \Rightarrow (in_E = false \rightarrow (out_i = in_{Fi})), \\ wcond(c) \Rightarrow (in_E = true \rightarrow (out_i = in_{Fi})), \\ wcond(c) \Rightarrow (in_E = false \rightarrow (out_i = in_{Ti})), \\ in_{Ti} = CEnv_1(s_i), \\ in_{Fi} = CEnv_2(s_i)\} \cup \{in_E = Top\}$$

    $\forall_{i=1}^{n} CEnv'(s_i) = out_i \wedge \forall_{s \notin \{s_1, \ldots, s_n\}} CEnv'(s) = CEnv(s)$

  - *Partial conversion*

    $conv(\texttt{if } E \texttt{ then } SB_1 \texttt{ else } SB_2 \texttt{ end if}, Env, CEnv, Comps, SD) =$
    $\quad (Env', CEnv', Comps', SD')$

    with

    $c = \texttt{if } E \texttt{ then } SB_1 \texttt{ else } SB_2 \texttt{ end if}, Env)$
    $(V, SD'', Top) = conv_E(E, Env, CEnv, Comps, SD)$

    We distinguish two cases

    1. $V = true$
       $(Env_1, CEnv_1, Comps_1, SD_1) = conv(SB_1, Env, CEnv, Comps, SD)$

$Env' = Env_1$
Let $s_1, \ldots, s_n$ be element of the set of all outputs of the conditional statement.
$SD' = SD \cup SD'' \cup SD_1 \cup$
$\bigcup_{i=1}^{n} \{ok(c) \Rightarrow (in_E = true \rightarrow (out_i = in_{Ti})), in_{Ti} = CEnv_1(s_i)\} \cup$
$\{in_E = Top\}$
$\forall_{i=1}^{n} CEnv'(s_i) = out_i \wedge \forall_{s \notin \{s_1,\ldots,s_n\}} CEnv'(s) = CEnv(s)$
$Comps' = Comps \cup Comps_1 \cup \{c\}$

2. $V = false$
$(Env_2, CEnv_2, Comps_2, SD_2) = conv(SB_1, Env, CEnv, Comps, SD)$
$Env' = Env_2$
Let $s_1, \ldots, s_n$ be element of the set of all outputs of the conditional statement.
$SD' = SD \cup SD'' \cup SD_2 \cup$
$\bigcup_{i=1}^{n} \{ok(c) \Rightarrow (in_E = false \rightarrow (out_i = in_{Fi})), in_{Fi} = CEnv_2(s_i)\} \cup$
$\{in_E = Top\}$
$\forall_{i=1}^{n} CEnv'(s_i) = out_i \wedge \forall_{s \notin \{s_1,\ldots,s_n\}} CEnv'(s) = CEnv(s)$
$Comps' = Comps \cup Comps_2 \cup \{c\}$

Note that full conversion of conditionals may be not computable because of a non-terminating branch, as in the following example:

```
...
T := 0; -- T < MAX
if ( T < MAX) then
     null;
else;
     while (T < MAX) loop
          null;
     end loop;
end if;
 ...
```

A solution to this problem is to finish conversion if to many components are created while converting the traced part of the conditional. This limit must be given by the user before conversion is started. In the case the limit is reached, only partial conversion is applied.

- *Loops* Loops are converted to a sequence of converted blocks. Apart from the case where the loop statement itself has to be removed to avoid faulty behavior, a loop itself only behaves incorrect because of a bug within the loop expression. This causes the loop statement to terminate too quickly or not quickly enough. Usually, the number of iteration steps expected and the number of iteration steps caused by (wrong) loops are nearly the same. We therefore use, as introduced for conditionals, fault modes. This fault modes are related to the difference of the iteration step counter. We assume the existence of a variable *deltaLoop* (given by the user) used for denoting fault modes regarding the iteration step counter. Note that fault modes leading to additional loops may lead to an non-terminating state. Consider for example the following program part:

```
...
T := 0;
while (T < MAX) loop
     ...
     while (T > MAX) loop
          null;
     end loop;
     T := T + 1;
end loop;
...
```

If we call the statement part of the enclosing loop statement with T > MAX we encounter the innermost loop statement which never terminates using the actual value for T. To avoid such problems we can (1) not use fault modes, (2) use fault modes only in cases where the problem does not occur, or (3) use fault modes only in cases where computation does not lead to an infinite loop. Case (1) has the big disadvantage that program diagnosis is limited. Case (2) leads to a syntactical description of programs where the problem does not occur. The problem never occurs if no loop statement is used in a loop, or if the innermost loops do not use variables within the termination condition which are influenced by statements of the outer loop. In case (3) we must be able to detect a termination or possible termination during conversion. This can be done by introducing a value limiting the maximum number of components created during conversion. If this number is reached the conversion of this loop is finished and the results are removed. No further conversion is done for this loop.

Note that in cases where the loop statement is only used for accessing array elements the number of loops is restricted by the array size. Furthermore, if a loop only ranges from the first array element to the last it is very unlikely that there is a fault within the termination condition of the loop statement. In this cases no fault modes are to be created.

For simplicity we introduce the conversion without taking account of the above described problems or limitations.

$inputs(\texttt{while } E \texttt{ loop } SB \texttt{ end loop}) = signals(E) \cup inputs(SB)$

$outputs(\texttt{while } E \texttt{ loop } SB \texttt{ end loop}) = outputs(SB)$

$$conv(\texttt{while } E \texttt{ loop } SB \texttt{ end loop}, Env, CEnv, Comp, SD) =$$
$$conv(\texttt{while } E \texttt{ loop } SB \texttt{ end loop}, Env, CEnv, Comp, SD, 0)$$

where

$$conv(\texttt{while } E \texttt{ loop } SB \texttt{ end loop}, Env, CEnv, Comp, SD, Index) =$$
$$(Env', CEnv', Comp', SD')$$

with

$(V, SD_E, Top) = conv_E(E, Env, CEnv, Comp, SD)$

$c = (\texttt{while } E \texttt{ loop } SB \texttt{ end loop}, Env)$

1. $V = true$ If the condition of the loop is true, then the statement part is evaluated.

$(Env'', CEnv'', Comp'', SD'') = conv(SB, Env, CEnv, Comp, SD, Index + 1)$
Let $s_1, \ldots, s_n$ be elements of the statement block's output.
$SD' = SD \cup SD_E \cup SD'' \cup \bigcup_{i=1}^{n} \{$
$ok(c, Index) \Rightarrow (out_i = in_{Ti}),$
$wbeh(c, Index) \Rightarrow (out_i = in_{Fi}),$
$in_{Ti} = CEnv''(s_i),$
$in_{Fi} = CEnv'(s_i)\} \cup \{Top = in_E\}$
$Comp' = Comp \cup Comp'' \cup \{c\}$
$\forall\, i \in \{1, \ldots, n\} : CEnv'(s_i) = out_i \wedge \forall\, s \notin \{s_1, \ldots, s_n\} : CEnv'(s) = CEnv''(s)$
$Env' = Env''$

2. $V = false$ If the condition is false, no further evaluation of the statement part is done. To introduce fault modes related to additional iteration steps, we further define:
$(Env_1, CEnv_1, Comp'_1, SD'_1) = conv(SB, Env, CEnv, Comp, SD)$
where
$SD_1 = SD'_1 \cup \{wbeh(c, Index + 1) \Rightarrow (out_i = in_{Ti}), ok(c, Index + 1) \Rightarrow (out_i = in_{Fi}), in_{Ti} = CEnv_1(s_i), in_{Fi} = CEnv(s_i)\}$
$Comp_1 = Comp'_1$
$\forall k \in \{2, \ldots, deltaLoops\}$:
$(Env_k, CEnv_k, Comp'_k, SD'_k) = conv(SB, Env_{k-1}, CEnv_{k-1}, Comp_{k-1}, SD_{k-1})$
where
$SD_k = SD'_k \cup \{wbeh(c, Index + k) \Rightarrow (out_i = in_{Ti}), ok(c, Index + k) \Rightarrow (out_i = in_{Fi}), in_{Ti} = CEnv_k(s_i), in_{Fi} = CEnv_{k-1}(s_i)\}$
$Comp_k = Comp'_k$
The system description, diagnosis components, and environments for the last iteration step can now be summarized as:
$SD' = SD \cup SD_E \cup$
$\{ok(c, Index) \Rightarrow (out_i = in_{Fi}), in_{Fi} = CEnv(s_i)\} \cup \{Top = in_E\} \cup$
$\bigcup_{k=1}^{deltaLoops} SD_k$
$Comp' = Comp \cup \{c\} \cup \bigcup_{i=1}^{deltaLoops} Comp_i$
$CEnv' = CEnv$
$Env' = Env$
The number of iterations is given by the function $index(c) = Index$.

As explained previously, *loops* have three basic fault modes *ok*, *wcond* and $undef$. They are defined by the following three rules which are assumed to be element of the system description for every *loop* statement.

$$ok(c) \rightarrow \left(\forall_{i=0}^{deltaLoops} ok(c, i)\right)$$

If we assume that we have done too many iteration steps, we can use the following fault mode:

$$wcond(c, Index) \rightarrow \left(\forall_{i=0}^{Index-1} ok(c, i) \wedge \forall_{i=Index}^{index(c)} wbeh(c, i) \wedge \forall_{i=index(c)+1}^{deltaLoops} ok(c, i)\right)$$

If too few iteration steps have been done, the following fault modes can be used:

$wcond(c, Index) \rightarrow \left(\forall_{i=0}^{index(c)} ok(c,i) \wedge \forall_{i=index(c)+1}^{Index} wbeh(c,i) \wedge \forall_{Index+1}^{deltaLoops} ok(c,i)\right)$

The above definitions allow to test faults regarding the number of iterations caused by the given loop condition. Faults related to wrong statements used in the loop that influence the number of iteration steps are not caught.

- *Null Statements* Null statements are ignored.

  $inputs(\texttt{null}) = \{\}$

  $outputs(\texttt{null}) = \{\}$

  $conv(\texttt{null}, Env, CEnv, Comps, Conn) = (Env, CEnv, Comps, Conn)$

- *Wait Statement* Wait statements are ignored.

  $inputs(\texttt{wait}\ldots) = \{\}$

  $outputs(\texttt{wait}\ldots) = \{\}$

  $conv(\texttt{wait}\ldots, Env, CEnv, Comps, Conn) = (Env, CEnv, Comps, Conn)$

4.1.2 Converting Expressions

Expressions used in statements are also converted to components with behavior.

- Expr = "E_1 `op` E_2"

  $conv_E(Expr, Env, CEnv, Comp, SD) = (op(V_1, V_2), Top, SD')$

  where $op \in \{and, or, xor, nand, nor, =, /=, <, <=, >, >=, +, -, *, /, \&, mod, rem, **\}$

  with

  $(V_1, SD_1, Top_1) = conv_E(E_1, Env, CEnv, Comp, SD)$

  $(V_2, SD_2, Top_2) = conv_E(E_2, Env, CEnv, Comp, SD)$

  $Top = out_{Expr}$

  $SD' = SD_1 \cup SD_2 \cup \{Top = op(in1_{Expr}, in2_{Expr}), in1_{Expr} = top_1, in2_{Expr} = top_2\}$

- Expr = "`si` E"

  $conv_E(Expr, Env, CEnv, SD) = (si(V), Top, SD')$

  where $si = \{+, -, abs, not\}$

  with

  $(V, SD'', Top'') = conv_E(E, Env, CEnv, SD)$

  $Top = out_{Expr}$

  $SD' = SD'' \cup \{Top = si(in_{Expr}), in_{Expr} = Top''\}$

- Expr = "`(` E `)`"

  $conv_E(Expr, Env, CEnv, SD) = conv_E(E, Env, CEnv, SD)$

- Expr = "$I($ $E_1, \ldots, E_n$ $)$"

  $conv_E(Expr, Env, CEnv, SD) = (V_T, SD', Top)$

  where I denotes a function.

  $\forall_{i=1}^{n}(V_i, SD_i, Top_i) = conv_E(E_i, Env, CEnv, SD)$

  $(V_T, SD_T, Top_T) = conv_F(FEnv(I), Env', CEnv', \bigcup_{i=1}^{n} SD_i)$

  with $Env', CEnv'$ defined only for the function arguments A_i such as follows: $Env'(A_i) = V_i$, $CEnv'(A_i) = Top_i$.

  $Top = out_{Expr}$

  $SD' = SD_T \cup \{Top = Top_T\}$

  The function body conversion is done like the usual conversion of sequential statements followed by the conversion of the return expression.

  $conv_E([St_1, \ldots, St_n, \texttt{return } E], Env, CEnv, SD) = (V, SD', Top)$

  where

  $(Env_S, CEnv_S, Comps_S, SD_S) = conv([St_1, \ldots, St_n], Env, CEnv, \{\}, SD)$

  $(V, SD', Top) = eval_E(E, Env_S, CEnv_S, SD_S \cup \bigcup_{C \in Comps_S}\{ok(C)\})$

  Note that this definition does not allow diagnosis inside a function call. To prevent this restrictions, all $ok(C)$ terms related to function body statements must be removed from the system description.

- Expr = "$I($ E $)$"

  $conv_E(Expr, Env, CEnv, SD) = (V, SD', Top)$

  where I denotes a signal or variable storing an array.

  $(V', SD'', Top') = conv_E(E, Env, CEnv, SD)$

  $(V_I, SD_I, Top_I) = conv_E(I, Env, CEnv, SD)$

  $V = V_I[V']$ Note that [ ] represents an array access to a signal or variable.

  $Top = out_{Expr}$

  $SD' = SD'' \cup SD_I \cup \{Top = Top_I[(in_{Expr})], in_{Expr} = Top'\}$

- Expr = "I" where $I \in CONST$

  $conv_E(Expr, Env, CEnv, SD) = (I, SD', Top)$

  with

  $Top = out_I$

  $SD' = SD \cup \{Top = I\}$

- Expr = "V" where $V \in VARS \cup SIG$

  $conv_E(Expr, Env, CEnv, SD) = (Env(V), SD, CEnv(V))$

4.1.3 Integrating Observations into SD_SEQ

Observations can be easily integrated into the previously defined context. From the conversion function, it follows that inputs of a program are automatically element of the system description. Outputs and other observations must be expressed in terms of **SD_SEQ**.

Observations are environments which are expected after statements have been executed. An observation can therefore be expressed by $obs(Env_{IN}, [St_1, \ldots, St_n], Env_{OUT})$. If only one statement is given, observations are usually given by $obs(Env_{IN}, St, Env_{OUT})$. While converting a program the following is done:

1. If St_1 should be converted, it must be proven that the given environment values (not equal to ϵ) stored in Env_{IN} are equal to the associated computed values. If this is not the case the observation is ignored.
2. Otherwise the observation is converted into a set $COBS$ after the conversion of St_n such that

   $\forall Env_{OUT}(X) \neq \epsilon : \{out_{(St_n, Env)} = Env_{OUT}(X)\} \in COBS$

   is true. Env denotes the environment computed by converting St_{n-1}.

The converted program SD_SEQ and observations $COBS$ can then be used for diagnosis as explained before.

4.1.4 Remarks on the Conversion of Programs

Before showing how the proposed conversion can be applied to program debugging it remains to prove that the conversion is correct wrt. the VHDL semantics. Fist we prove that the expression conversion is correct. This can be shown as follows:

Theorem 4.1.1 (Correctness of Expression Conversion) *Let E be an expression, Env an environment including values for signals and variables and $(Val, Top, SD') = conv_E(E, Env, CEnv, Comp, SD)$ the result of applying the conversion function to E. We assume that $SD \cup \{ok(c) | c \in Comp\} \models CEnv(V) = Env(V)$. The conversion function is correct wrt. to the expression evaluation, i.e., $eval(E, Env) = Val$ and $SD' \models eval(E, Env) = Top$.*

> **Proof:** The first part of the theorem is trivially true. The second part must be proved by induction over the size of the expression. For the first part of the induction proof (1) we assume that the expression size is equal to 1. The expression therefore has the form I or V where I denotes a constant and V a signal or variable. Since $conv_E(I, Env, CEnv, Comp, SD) = (I, out_I, SD \cup \{out_I = I\})$ and $eval(I, Env) = I$, the theorem is obviously true for this case. The same holds for $conv_E(V, Env, CEnv, Comp, SD) = (Env(V), SD, CEnv(V))$ because of the initial assumption. In the second part of the proof (2) we assume that the theorem holds for expression of size $\leq n$, and prove the induction hypothesis in the final part (3) for

expression with a size $n+1$. Since the conversion function does the same as the evaluation function the induction step must also deliver the expected result. To complete the proof, we show this in the case where the expression has the form $E_1 op E_2$. The other cases are analogous. From $conv_E(E_1 op E_2, Env, CEnv, Comp, SD) = (op(E_1, E_2), out_E, SD \cup SD_1 \cup SD_2 \cup \{out_E = op(in_1, in_2), in_1 = Top_1, in_2 = Top_2\}$ and the induction hypothesis follows: $SD_1 \cup SD \models in_1 = eval(E_1, Env)$ and $SD_2 \cup SD \models in_2 = eval(E_2, Env)$. So $SD \cup SD_1 \cup SD_2 \models out_E = eval(E_1 op E_2, Env)$ must also be true if the *op* function is correct. ■

The theorem about the correctness of expression conversion is used in the proof of the theorem stating the correctness of **SD_SEQ**.

Theorem 4.1.2 (Correctness of SD_SEQ) *Let P be a process and conv the conversion function described above. The conversion function is correct wrt. the VHDL semantics of P, if $(Env', CEnv, Comps, SD) = conv(P, Env)$ is the result of the conversion and $Env' = eval(P, Env)$ the result of the process evaluation, then it holds that:*

$$\Big(\forall C \in Comps : OK(C)\Big) \Rightarrow \Big(\forall S \in DRIVERS \cup VARS : SD \models Env'(S) = CEnv(S)\Big)$$

Proof: In this proof we use the result of theorem 4.1.1 showing that expressions are converted in the expected manner. Assume that all diagnosis components behave correct, i.e., $\forall C \in Comps : OK(C)$. We prove the theorem by induction over the number of statements and by simply comparing the rules for evaluation and conversion. Without restricting generality, we assume that at most one signal assignment for every signal is used in the process.

(1) The induction starts with processes including one statement. Since every process must include a *wait* statement at the end and the a wait statement has no direct influence on signal values the theorem is obviously true in this case.

(2) The same holds for processes including two statements. In this case the first statement must be a *null* statement, a *variable assignment*, or a *signal assignment*. In the first case the theorem is trivially true. In both other cases the theorem is true, too. The semantics of an assignment statement is to change the environment or the projected output waveform of the target. In case of a *variable assignment* the environment is changed using the expression value, so $Env'(T)$ is determined by the expression used. The same propositions can be drawn for the converted statement, since the system description has the form $ok((St, Env)) \Rightarrow (out_{(St,Env)} = Top)$ where Top denotes the top component associated with the expression. By assuming that expressions are correctly converted, the theorem must hold. Finally, we must prove the same for *signal assignments*. Since only one value can be result of the process (because of the chosen assumptions) we must only prove that this value is equal to the value of the converted process. Using the same arguments as for *variable assignments* ensures this.

From now we assume that the theorem is correct for processes with at most n statements, and prove that it still remains correct for $n+1$ statements. If the new statements is a *null*, *variable assignment*, or *signal assignment* statement, this is obviously true. It still remains to prove that it also holds for *conditionals* and *loops*.

1. *Conditionals* Because of the used assumptions, the proof for partial and complete conversion is the same. Without restricting generality, we show that the theorem is true by assuming that the conditional expression is true.

   By the assumption that expressions are correctly converted and the conversion is correct for the processes with less than $n + 1$ statements the conditional expression must have the associated correct value. Because of the assumption that the expression is true, we must only show that the theorem holds for all variables and signals used as output of the first statement block. This must be the case because of the induction hypothesis.

2. *Loops* Using similar arguments as for *conditionals* the correctness can also be shown in this case. For simplicity we assume that *deltaLoops* is equal to 0. Since only the *ok* mode is of interest, this assumption does not influence the proof. An additional argument is that the number of steps used for evaluation of the loop must be equal to the number of inputs. Which is obviously true.

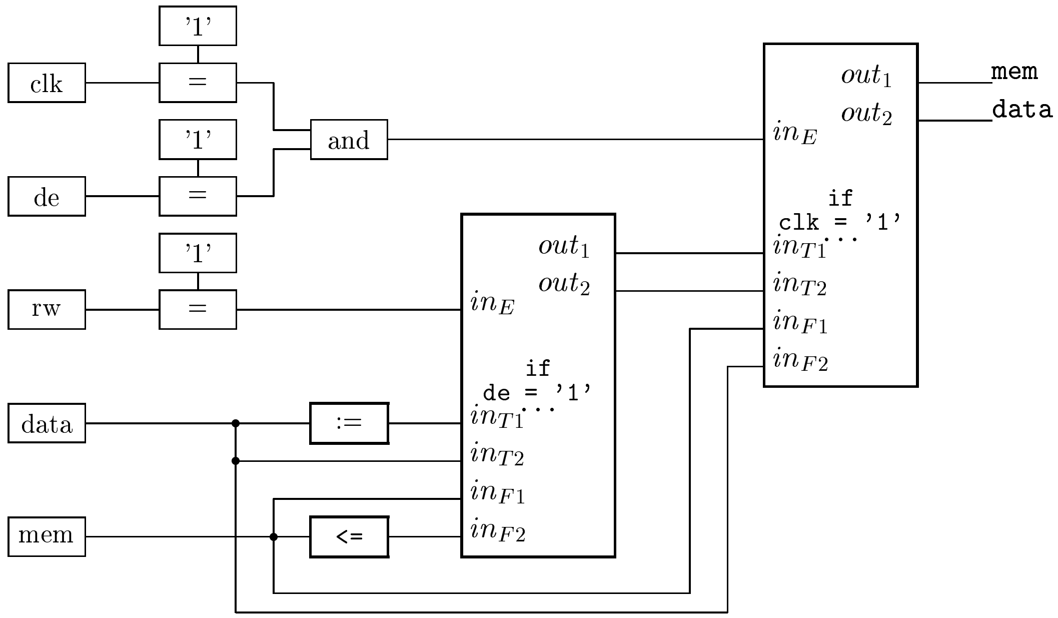
■

We now have proved that the conversion of programs (exactly processes) to trace evaluation graphs is correct wrt. the program evaluation.

4.1.5 A short Conversion Example

We use example 1.1 to show how programs (processes) are converted to diagnosis components. Consider further that the starting environment is given by $Env(clk) =' 1'$, $Env(de) =' 1'$, $Env(rw) =' 0'$, $Env(mem) ='' 0010''$, $Env(data) ='' UUUU''$, and that architecture `behav` is to be converted.

In the first step of converting the process `MEM_UNIT` the initial values associated to signals and variables are converted to constant components which are not used for diagnosis. Then the first conditional is converted by converting the expression and both statement blocks. The conversion of the expression is done using the conversion rules for expression. This leads to an *and* component connected to two *equal* components. Conversion of the second conditional concludes computation. After this the diagnosis system looks like follows:



The above converted diagnosis system does not include a component for the *wait* statement since they are ignored. The system description related to the component oriented sight is given by:

$c1 = ($`if clk = '1' ...`$, Env)$

$c2 = ($`if de = '1' ...`$, Env)$

$c3 = ($`mem := data`$, Env)$

$c4 = ($`data <= mem`$, Env)$

$c_{clk}, c_{de}, c_{rw}, c_{mem}, c_{data}$ denotes the components related to the initial signal values.

$SD = \{out(c_{clk}) =' 1', out(c_{de},' 1'), out(c_{rw}) =' 1', out(c_{mem}) ='' 0010'', out(c_{data}) ='' UUUU'', \ldots, out_{de='1'} = equal(in1_{de='1'}, in2_{de='1'}), in1_{de='1'} = out_{de}, in2_{de='1'} =' 1', \ldots\}$

The system description part for component $c1$ for example is given by:

$ok(c1) \Rightarrow (in_E(c1) = true \rightarrow (out_1(c1) = in_{T1}(c1)))$

$ok(c1) \Rightarrow (in_E(c1) = true \rightarrow (out_2(c1) = in_{T2}(c1)))$

$ok(c1) \Rightarrow (in_E(c1) = false \rightarrow (out_1(c1) = in_{F1}(c1)))$

$ok(c1) \Rightarrow (in_E(c1) = false \rightarrow (out_2(c1) = in_{F2}(c1)))$

$wcond(c1) \Rightarrow (in_E(c1) = true \rightarrow (out_1(c1) = in_{F1}(c1)))$

$wcond(c1) \Rightarrow (in_E(c1) = true \rightarrow (out_2(c1) = in_{F2}(c1)))$

$wcond(c1) \Rightarrow (in_E(c1) = false \rightarrow (out_1(c1) = in_{T1}(c1)))$

$wcond(c1) \Rightarrow (in_E(c1) = false \rightarrow (out_2(c1) = in_{T2}(c1)))$

The system description parts of the other components can be easily obtained using the informations provided in the conversion function.

4.2 Using SD_SEQ for Diagnosis

The conversion example of subsection 4.1.5 has shown how processes can be converted into a related system description which can be used for program debugging. Consider for example that the output value "0011" is expected for the signal `data`. This can be expressed by the first order fact $out_1(c2) =" 0011"$. Assuming that all four components behave correct and including the observation leads to the derivation of an inconsistency which can be easily seen. Assuming that $c1$ is a single diagnosis using the fault mode *wcond* still leads to an inconsistency. Only the single diagnoses $undef(c1), undef(c2)$ and $undef(c4)$ can explain the faulty behavior. Since $undef(c1)$ and $undef(c2)$ are not very likely if we assume that changing the program's structure is very unlikely, the only remaining canditate of interest is $undef(c4)$, which is indeed a very good hint for a programmer to look at.

To formalize the above ideas, we can use the fault probabilities of components and their modes. As introduced in section 1.2.1 the probability $p_m(c)$ gives the fault probability of mode m and component c. By using the following table

Component	*ok*	*undef*	*wcond*
c1	0.79	0.01	0.2
c2	0.79	0.01	0.2
c3	0.7	0.3	
c4	0.7	0.3	

the diagnosis probabilities can be computed.

$p(\{undef(c1)\}) = 0.387$

$p(\{undef(c2)\}) = 0.387$

$p(\{undef(c4)\}) = 0.13$

The advantage of using probabilities to focus on special diagnoses is their scalability. The fault probabilities can be collected for VHDL (or other) programming languages. The diagnosis accuracy can then be tuned by changing the probability values for every programmer or software project dynamically.

In the rest of this section, we summarize the semantics behind the fact that a software part, i.e., a statement fails. We will use this information in later sections to improve the system description.

- *Signal Assignment*

  Saying that a signal assignment fail leads to several repair actions. First it is possible that the expression has a fault. To find such faults we can extend the system description by converting expressions and expression parts to diagnosis components. This leads to a larger number of diagnosis components and an increasing computation time. Secondly the target can be the bug. Replacing the target by another signal may lead to a correct result. On the other hand the statement itself should be removed or converted to a variable assignment (while using a variable instead of a signal as target). As the last possibility related to an unknown bug in a signal assignment is that the statement is correct but wrongly placed within the statement part. The **SD_SEQ** system description does not allow to differentiate between this possible faults.

Note that this enumeration of possible faults does not include timing faults since they are not in the focus of this work.

- *Variable Assignment*

  For variable assignments the same as for signal assignments can be said.

- *Conditional*

  Conditional statements may cause a bug because of several reasons. First the statement itself might have to be removed, the statement block of the *if* branch might have to be replaced by the block of the *then* branch, or the expression is buggy. It can be easily seen that a change of the statement blocks can be expressed by negating the expression's value. To allow expressing that the condition's expression behaved incorrectly the fault mode *wcond* has been introduced. The remaining possible faults (removing the conditional or replacing) it are indicated by the $undef$ mode.

- *Loop Statement*

  The loop statement may fail in a manner similar to the conditional because it is unnecessary, because it should be replaced by statements, or because the expression used to control the number of loops has a bug. The first two fault modes are expressed by the $undef$ mode. No further differentiation between them is possible within **SD_SEQ**. The last possible fault mode is expressed by the mode *wcond* with the number of loops as parameter. A diagnosis for example saying that the condition should be changed to extend the number of loops by 2 not automatically give the solution of how to correct the bug. On the other hand, this knowledge is may be not enough to fix the bug because, for example, it can be also fixed by changing a statement within the loop block to increase certain values. However knowing that the loop may terminate too early or too late can help the programmer in correcting the bug.

The introduced model for diagnosis of sequential software programs is not complete at all. Faults related to missing program parts cannot be found or cause misleading diagnoses. The first case occurs when a signal (variable) is said to behave incorrectly and is not changed by a statement. In this case no diagnoses can be computed. To avoid the problem, the components c_s can be inserted into the set of diagnosis components for all signals and variables s. This solution increases the number of diagnoses. By associating a very small probability with the component the diagnosis related to the signal components will be considered very unlikely.

Chapter 5

Software Debugging using Replacements

5. Der Satz ist eine Wahrheitsfunktion der Elementarsätze.
(Der Elementarsatz ist eine Wahrheitsfunktion seiner selbst.)

Ludwig Wittgenstein, 1921
Tractatus logico-philosophicus
Logisch-philosophische Abhandlung

This chapter describes the system description **SD_ER** using expression replacements to improve the diagnosis process. The presented model is an extension to [SW96b, SW96a] where the idea of using replacements has been introduced[1]. In difference to previous work this chapter provides a model for a more extended subset of VHDL. The way of using the system description **SD_SEQ** to allow focusing on relevant diagnosis components and therefore improve diagnosis is added to this chapter, too.

In difference to **SD_SEQ** we do not use the evaluation trace for diagnosis. Instead we introduce a new evaluation function using fault models to guide result computation. The fault modes allow to express possible replacements which are then used while program execution is done. Similar to the other approaches the original program is converted to a diagnosis system. During conversion possible replacements are computed based on the assumption that the correct program is only a small variant of the existing one. To avoid the problem that no diagnoses can be computed using replacements an $undef$ fault mode is introduced and indicates that the used replacements are not sufficient to explain a faulty behavior. As mentioned previously, the introduced $undef$ mode is also not enough to deal with missing functionality. A solution to this problem that does not require changing the basics of the introduced approach can be found in chapter 6.

The main problem arising when using replacements is the possibility of introducing an expression that leads to a non-terminating program. As a simple but effective solution, we assume the

[1] The addition and removal of program parts to ensure correctness was also used in [CFD93] for logic programs.

existence of a (given) limit (time limit or limiting maximum number of iteration steps). It is further assumed that the violation of this limit causes a program evaluation error indication. Because we are using a consistency based approach, we formalize this error by allowing to derive a contradiction. Note that this approach is not a general one. It is always possible that the given limit is to strong causing a loss in possible program error explanations. However, we use the argument of searching for a new program, which is only a small variant of the original one, and extend it by searching for a program with an almost equal evaluation behavior with regard to resources (time, space).

The termination problem of expression replacements is not a severe one for all programs. Programs that do not contain recursive functions (as programs written in **RVHDL**) or loops always terminate. If there is a loop used within a program, a replacement also does not change the termination behavior if it only influences variables (or signals) not influence (explicitly or implicitly) the loop condition.

Example 10: The process

```
P1:  process
begin
     S <= not I;
     wait on I;
end process P1;
```

cannot cause an infinite loop. The termination behavior does not depend on an expression replacement for `not I`. On the other hand, for the process

```
P2:  process
     variable I: integer := 0;
begin
     I := 10
     while I > 0 loop
          ...
          I := I - 1;
     end loop;
end process P2;
```

the (possible) replacement `I` for `I - 1` causes a never terminating loop.

□

Theorem 5.0.1 *Let P be a process and $E \leftarrow E'$ an expression replacement. The expression replacement never changes the termination behavior if one of the following rules is fulfilled:*

1. *P contains no loop.*

2. *If P contains a loop where $S_1, \ldots, S_n$ are variables (signals) used in the loop condition, then the replacement is termination neutral iff E does not have an influence of the values of $S_1, \ldots, S_n$.*

The proof is omitted because of simplicity. If a process contains no loops it allways terminates. If the loop condition is not influenced by the replacement the replacement, must be termination neutral.

In the following sections we introduce how expressions can be replaced by other expressions and how this concept can be integrated into the existing model-based approach. A formal model for VHDL is presented and explained using an example. Additional problems caused by introducing automatic programming techniques are pointed out. The problems remain unsolved and are topics for further research.

5.1 Computing Expression Replacements

In this section we introduce an algorithm delivering evalution different expressions for a given expression. The maximum difference between the original expression and a possible replacement expression is assumed to be specified before using the algorithm. The restriction to new expressions not evaluating to the same result as the original one naturally arises because we want to explain misbehavior by showing how to correct it.

While computing replacements, the knowledge about functions, constants, signals, and variables which can be used must be provided. This is done by introducing a data structure storing functions with their types called function type environement $FTEnv$, a set SIG and $VARS$, and a function *type* associating variables and signals to their type. A function *domain* associating types to values is also assumed to be available.

Definition 5.1.1 (Function Type Environment) *A function type environment $FTEnv$ is a set of triples (F, T, TS) where F denotes the name of a function, T the type of the resulting value, and TS a sequence of types $< t_1, \ldots, t_n >$ related to arguments $< E_1, \ldots, E_n >$.*

Operations related to function type environments are:

type : $FUNC \times 2^{FEnv} \mapsto TYPES$ answers the type of the result value and is defined by $type(F, FTEnv) = T$ iff $\exists\, (F, T, TS) \in FTEnv$.

$functA : 2^{FTEnv} \times \mathbb{N} \mapsto 2^{\mathrm{FTEnv}}$ returns new function environment consisting only out of functions of the given arity and is defined by $functs(FTEnv, N) = \{(F, T, TS) | (F, T, TS) \in FTEnv, |TS| = N\}$.

$functT : 2^{FTEnv} \times TYPES \mapsto 2^{FTEnv}$ returns a function environment consisting only out of functions having the given result type. $functT(FTEnv, T) = \{(F, T, TS) | (F, T, TS) \in FTEnv\}$

$functTS : 2^{FTEnv} \times 2^{TYPES} \mapsto 2^{FTEnv}$ returns a function environment consisting only out of functions having the given argument types. $functT(FTEnv, TS) = \{(F, T, TS) | (F, T, TS) \in FTEnv\}$

Initially we assume that the function type environment has at least the following elements:

1. For $f \in \{and, or, xor, nand, nor\}$:
   $(f, bit, < bit, bit >)$
   $(f, std\_logic, < std\_logic, std\_logic >)$,
   $(f, std\_logic\_vector, < std\_logic\_vector, std\_logic\_vector >)$
2. For $f \in \{not\}$:
   $(f, bit, < bit >)$,
   $(f, std\_logic, < std\_logic >)$,
   $(f, std\_logic\_vector, < std\_logic\_vector >)$
3. $(\&, std\_logic\_vector, < std\_logic\_vector, std\_logic >)$,
   $(\&, std\_logic\_vector, < std\_logic, std\_logic >)$,
   $(\&, std\_logic\_vector, < std\_logic, std\_logic\_vector >)$,
   $(\&, std\_logic\_vector, < std\_logic\_vector, std\_logic\_vector >)$
4. For $f \in \{=, / =\}$:
   $(f, boolean, < bit, bit >)$,
   $(f, boolean, < std\_logic, std\_logic >)$,
   $(f, boolean, < std\_logic\_vector, std\_logic\_vector >)$

Computing replacements is restricted by the given difference between original expression and the expression searched for. It is further assumed that functions which are possible replacements must have the same argument types as the function of the original expression, and that the number of arguments of a function does not change. Note that this restrictions can be easily removed.

Algorithm REPLACEMENTS(E, Env,FTEnv, D)

Answers a set of expression evaluating to different values where the difference between E and every other expression is less or equal to D.

1. Call the subprocedure REPLACE(E,Env,FTEnv,D) and prune the result by removing those elements evaluating to the same value as the expression E.
2. Return the resulting list after pruning.

Algorithm REPLACE(E,Env,FTEnv,D)

1. If $D = 0$ return a set with the single element E as result. Otherwise the result set is defined as follows:

   (a) If E has the form "E_1 `op` E_2" then the result set is defined by

   $$\begin{aligned}&\{E' | E' \in REPLACE(E_1, Env, FTEnv, D - depth(E_2) - 1) \vee \\ &\qquad E' \in REPLACE(E_2, Env, FTEnv, D - depth(E_1) - 1)\} \cup \\ &\{f(E_1', E_2') | f = functT(functTS(FTEnv, < type(E_1), type(E_2) >), type(si)) \wedge \\ &\qquad\qquad E_1' \in REPLACE(E_1, Env, FTEnv, D_1) \wedge \\ &\qquad\qquad E_2' \in REPLACE(E_2, Env, FTEnv, D_2) \wedge D_1 + D_2 = D - 1\}.\end{aligned}$$

   (b) IF E has the form "`si` E_1" then the result set is defined by

   $$\begin{aligned}&\{E' | E' \in REPLACE(E_1, Env, FTEnv, D - 1)\} \cup \\ &\{f(E') | f = functT(functTS(FTEnv, < type(E_1) >), type(si)) \wedge \\ &\qquad E' \in REPLACE(E_1, Env, FTEnv, D - 1)\}.\end{aligned}$$

   (c) If E has the form "`(` E_1 `)`" the result set is given by a recursive function call $REPLACEMENTS(E_1, Env, FTEnv, D)$.

   (d) If E has the "I`(` E_1 `)`" where I denotes an array access the result set is given by $\{I(E') | E' \in REPLACE(E_1, Env, FTEnv, D)\}$.

   (e) If E has the form "I`(` $E_1, \ldots, E_n$ `)`" where I denotes a function the result set is given by

   $$\begin{aligned}&\{E' | E' \in REPLACE(E_1, Env, FTEnv, D - 1)\} \cup \\ &\{f'(E') | f' = functT(functTS(FTEnv, < type(E_1) >), type(I)) \wedge \\ &\qquad E' \in REPLACE(E_1, Env, FTEnv, D - 1)\}.\end{aligned}$$

   (f) If E denotes a constant "I" then the result set is given by

   $$\begin{aligned}&\{I' | I' \in CONST \cup VARS \cup SIG\} \cup \\ &\{f(E') | f = functT(funcTS(FTEnv, < type(I) >), type(I)) \wedge \\ &\qquad E' \in REPLACE(I, Env, FTEnv, D - 1)\}\end{aligned}$$

   (g) If E denotes a variable or signal "V" then the result set is given by

   $$\begin{aligned}&\{V' | V' \in CONST \cup VARS \cup SIG\} \cup \\ &\{f(E') | f = functT(funcTS(FTEnv, < type(V) >), type(V)) \wedge \\ &\qquad E' \in REPLACE(V, Env, FTEnv, D - 1)\}\end{aligned}$$

2. Return the expression set as result.

The complexity of the algorithm can be expressed in terms of the number of functions, the expression, and the difference parameter D.

Theorem 5.1.1 (Time Complexity of REPLACEMENTS) *The time complexity of REPLACEMENTS is of order $o(|FUNC|^D)$ where $FUNC$ denotes the set of used functions stored in the function environment and D the difference between original expression and result expression.*

Proof: The time complexity of REPLACEMENT is given by the time complexity of REPLACE. To give an estimate for this complexity we assume that only expression creation is time intensive.

Since we are only interested in the worst case time complexity, we only look at two cases.

1. Assume that the expression depth is equal to 1. In this case the expression E is either a constant, variable or signal. In this case the time complexity is given by creating all possible replacements.
   $TC(D) = |CONST \cup VARS \cup SIG| + |FUNC| \cdot TC(D-1)$
   Because $TC(D-1)$ can be computed equal to $TC(D)$ we get as result:
   $TC(D) = \sum_{i=0}^{D} |FUNC|^i \cdot |CONST \cup VARS \cup SIG|$
   which has the order $o(|FUNC|^D)$.
2. If the expression depth is greater than one the worst case time complexity is given by using rule 1a which can be expressed by:
   $TC(D) = TC(D-1-depth(E_1) + TC(D-1-depth(E_2)) + |FUNC| \cdot (\sum_{D_1+D_2=D-1} TC(D_1) + TC(D_2))$.
   It is obvious that in this case $TC(D) \sim o(|FUNC|^D)$ is true.

■

Because time complexity and the number of computed expressions are of order $o(|FUNC|^D)$ it is very important not to use to large values for D. As we will see in the rest of the chapter the number of fault modes depends also on the number of different replacements. Since the number of fault modes have also a great influence on the overall diagnosis time, the number of replacements should not be too high.

A final example shows how all possible expression replacements are computed.

Example 11: Consider the expression `S1 or (not S2)` where `S1,S2` denotes two signals of type std_logic, and let the environment be $Env(S1) =' 1'$, $Env(S2) =' 0'$. Assume further that the function environment contains only the functions `or, and, not, xor, nand, nor`.

In the first step the function call REPLACE(S1 or (not S2), Env, FEnv, 2) delivers the following results:

$$
\begin{aligned}
&\{E' | E' \in REPLACE(S1, Env, FEnv, 1) \vee \\
&\qquad E' \in REPLACE((notS2), Env, FEnv, 1)\} \cup \\
&\{E'_1 op E'_2 | E'_1 \in REPLACE(S1, Env, FEnv, 1) \wedge \\
&\qquad\qquad E'_2 \in REPLACE((notS2), Env, FEnv, 0) \wedge \\
&\qquad\qquad op \in \{or, and, xor, nand, nor\}\} \cup \\
&\{E'_1 op E'_2 | E'_1 \in REPLACE(S1, Env, FEnv, 0) \wedge \\
&\qquad\qquad E'_2 \in REPLACE((notS2), Env, FEnv, 1) \wedge \\
&\qquad\qquad op \in \{or, and, xor, nand, nor\}\}
\end{aligned}
$$

In the second step we compute REPLACE(S1, Env,FEnv,1) delivering the results:

$\{S1, S2, '0', '1', '-', 'H', 'L', 'U', 'Z', 'W', 'X'\}$

and

$\{not(E') | E' \in REPLACE(S1, Env, FEnv, 0)\} = \{not(S1)\}$

In the third step the recursive call REPLACE((not S2), Env, FEnv, 1) is done delivering:

$\{not(S2), S2, not(S1), not('0'), not('1'), \ldots\}$

This concludes the original call to the REPLACE function. The REPLACEMENT function calling REPLACE will remove expressions that do not provide a new value. The final result is:

$\{notS1, S2, (notS1)and(notS2), S1xorS2, S1nor(notS2),$
$(notS1)nor(notS2), S1norS2, S1nand(notS1), \ldots\}$

including also replacements which introduce constants.

□

Since it is very unlikely that variables and signals are to be replaced by constant values, we introduce the concept of expression generality indicating whether an expression is more complex than another. For this purposes we focus only on the most complex expressions.

Definition 5.1.2 (More Complex Expression) *Let E_1, E_2 be two expressions. We say that E_1 is more complex than E_2 iff*

1. *The number of variables and signals used in E_1 is greater than the number of variables and signals used in E_2.*
2. *If the number of variables is equal in both expressions than the expression size (the overall number of subexpressions) of E_1 must be greater than for E_2*

$moreComplex(E_1, E_2) \Leftrightarrow (signals(E_1) > signals(E_2)) \vee (signals(E_1) = signals(E_2) \rightarrow |E_1| > |E_2|$

5.2 Introducing SD_ER

As mentioned above we introduce expression replacements to improve the diagnosis process by reducing the number of diagnoses. The integration of expression replacements to basic model-based diagnosis techniques is done using fault modes (with associated models). For the sake of simplicity and to allow a comparison between **SD_SEQ** and **SD_ER** we concentrate only on the VHDL sequential part. However the introduced concepts can be also easily integrated into full **RVHDL** .

The fault modes used can be summarized as:

1. *General Statements:* For all sequential statement types used in **RVHDL** , the following fault modes are relevant:

   Correct behavior (*ok*) Correct behavior causes the computation of the statement behavior using the program semantics as given.

   Spurious statement (Remove statement) (*remove*) The effect of removing a statement is to leave variable and signal values unchanged during execution.

   Wrong statement (Replace statement by one or more new statements) (*replace*) Unfortunately, the task of finding correct replacements for a complete statement in general amounts to nothing short of automatic programming [WL69]. We therefore have to artificially restrict the search space which is done by limiting repair size.

   Undefined fault mode (*undef*) The undefined failure mode is only used if all other modes cannot explain the misbehavior. It does not try to explain variable or signal values, and would cover cases where the program contains deep-seated logical errors.

2. *Signal and Variable Assignment:* These two assignment types largely agree in the possible modes.

   Wrong target (*target*) If a wrong target is assumed, we can check it by inserting a new one. Only defined variables (signals for signal assignments) can be used for target replacement.

   Wrong expression (*expr*) A wrong expression implies that there must be a correct one. Checking this mode is done by replacing the original expression by a new one. The type of the result must be the same.

   Wrong time expression (*time*) This mode exists for signal assignment statements only and indicates an error in the `after` clause.

3. *Conditional Statement:* The conditional statement can fail using the following special mode:

   Wrong condition (*cond*) This mode specifies that the condition of the conditional statement is wrong and should be changed to the given new expression.

   In difference to [SW96a] the conditional mode does not indicate that the whole condition is buggy for all environments. The condition may fail only for one environment. Therefore trying to replace the old condition by a new one is the correct fault mode for conditionals. However in the case where loops are not considered the model described in [SW96a] is sufficient.

4. *Loops Statement:* The loop statement can fail like a conditional statement:

   Wrong condition (*cond*) This mode specifies that the condition of the loop statement is wrong and that it should be replaced by a given new expression.

The above basic fault modes must be related to formal models expressing the semantics. As mentioned above, the case where a statement should be replaced by another statement or statement list is omitted and therefore an open point of research. The following rules describing the semantics of the different statement modes are assumed to be element of the system description **SD_ER**. This can be expressed as usual by introducing a conversion function. The conversion function $conv_{ER}$ uses the process as input and converts it to a system description and diagnosis statements.

Assume a process P of the form Id `process` PD `begin` $St_1 \dots St_n$ `end process` Id.

$conv_{ER}(P) = conv_{ER}(St_1; \dots St_n; \{\}, SD) = (Comps', SD')$

Blocks are converted into components by sequentially converting each statement into a component.

$$
\begin{aligned}
& conv_{ER}(St_1; \dots St_n; , Comps, SD) = \\
& \quad conv_{ER}(St_2; \dots ; St_n; , Comps', SD') \\
& \qquad \text{with } (Comps', SD') = conv(St_1, Comps, SD)
\end{aligned}
$$

Every statement St is converted by

$conv_{ER}(St, Comps, SD) = (Comps \cup \{St\}, SD \cup SD(St)$.

The behavior $SD(St)$ stored in the system description for a signal is defined as follows.

If a component behaves correctly we can use the original VHDL semantics for sequential statements described in section 2.2.

- *Signal Assignments*
  - S = "T`<=`E `after` TE"

    $$
    \begin{aligned}
    & ok(S) \Rightarrow (eval(S, Env, Pow) = \\
    & \quad (Env, (Pow(T))[(eval_E(E, Env), ActualTime + eval_E(TE, Env)]_{INERT}))
    \end{aligned}
    $$

  - S = "T`<= transport` E `after` TE"

    $$
    \begin{aligned}
    & ok(S) \Rightarrow (eval(S, Env, Pow) = \\
    & \quad (Env, (Pow(T))[(eval_E(E, Env), ActualTime + eval_E(TE, Env)]_{TRANS}))
    \end{aligned}
    $$

- *Variable Assignments* S ="$T := E$"

  $ok(S) \Rightarrow (eval(S, Env, Pow) = (Env[(T, eval_E(E, Env))], Pow))$

- *Conditionals* S = "`if` E `then` SB_1 `else` SB_2 `end if`"

  $$
  \begin{aligned}
  & ok(S) \Rightarrow (eval_E(E, Env) = true \rightarrow \\
  & \qquad eval(S, Env, Pow) = eval(SB_1, Env, Pow))
  \end{aligned}
  $$

  $$
  \begin{aligned}
  & ok(S) \Rightarrow (eval_E(E, Env) = false \rightarrow \\
  & \qquad eval(S, Env, Pow) = eval(SB_2, Env, Pow)
  \end{aligned}
  $$

- *Loops* S = "`while` E `loop` SB `end loop`"

$$
\begin{aligned}
ok(S) \Rightarrow (&eval_E(E, Env) = true \rightarrow \\
&\quad eval(S, Env, Pow) = eval(S, Env', Pow')) \\
&\qquad \text{where } (Env', Pow') = eval(SB, Env, Pow)
\end{aligned}
$$

$$
\begin{aligned}
ok(S) \Rightarrow (&eval_E(E, Env) = false \rightarrow \\
&\quad eval(S, Env, Pow) = (Env, Pow))
\end{aligned}
$$

- *Null Statements*

  $ok(\texttt{null}) \Rightarrow (eval(\texttt{null}, Env, Pow) = (Env, Pow))$

- *Wait Statement*

  $ok(\texttt{wait}\ \ldots) \Rightarrow (eval(\texttt{wait}\ \ldots, Env, Pow) = (Env, Pow))$

Removing a statement without replacing it can be done by not changing the environment and the projected output waveform. This fault mode has the following associated model.

$remove(S) \Rightarrow eval(S, Env, Pow) = (Env, Pow)$

Undefined behavior can be expressed by removing variables and signals from the actual environment and projected output waveform. Variables and signals to be removed must be used as target within the statement which is assumed to behave undefined.

- *Signal Assignments*

  – S = "T`<=`E `after` TE"

  $$
  \begin{aligned}
  undef(S) \Rightarrow (&eval(S, Env, Pow) = \\
  &(Env, Pow - T)
  \end{aligned}
  $$

  – S = "T`<= transport` E `after` TE"

  $$
  \begin{aligned}
  undef(S) \Rightarrow (&eval(S, Env, Pow) = \\
  &(Env, Pow - T)
  \end{aligned}
  $$

- *Variable Assignments* S ="T := E"

  $undef(S) \Rightarrow (eval(S, Env, Pow) = (Env - T, Pow))$

- *Conditionals* S = "`if` E `then` SB_1 `else` SB_2 `end if`"

$$
\begin{aligned}
undef(S) \Rightarrow (&eval_E(E, Env) = true \rightarrow \\
&\quad eval(S, Env, Pow) = (Env'', Pow'') \\
&\qquad \text{where } (Env', Pow') = eval(SB_1, Env, Pow)) \text{ and} \\
&\qquad Env'' = (\ldots(Env' - V_1) - \ldots) - V_n \text{ with } V_i inoutputs(SB_1) \wedge V_i \in VARS \\
&\qquad Pow'' = (\ldots(Pow' - S_1) - \ldots) - S_m \text{ with } S_i \in outputs(SB_1) \wedge S_i \in SIG
\end{aligned}
$$

$undef(S) \Rightarrow (eval_E(E, Env) = false \rightarrow$
$\quad eval(S, Env, Pow) = (Env'', Pow'')$
$\quad\quad$ where $(Env', Pow') = eval(SB_2, Env, Pow))$ and
$\quad\quad Env'' = (\ldots(Env' - V_1) - \ldots) - V_n$ with $V_i inoutputs(SB_2) \wedge V_i \in VARS$
$\quad\quad Pow'' = (\ldots(Pow' - S_1) - \ldots) - S_m$ with $S_i \in outputs(SB_2) \wedge S_i \in SIG$

- *Loops* S = "`while` E `loop` SB `end loop`"

  $undef(S) \Rightarrow (eval(S, Env, Pow) = (Env'', Pow''))$
  $\quad\quad Env' = (\ldots(Env - V_1) - \ldots) - V_n$ with $V_i inoutputs(SB) \wedge V_i \in VARS$
  $\quad\quad Pow' = (\ldots(Pow - S_1) - \ldots) - S_m$ with $S_i \in outputs(SB) \wedge S_i \in SIG$

- *Null Statements*

  $undef(\texttt{null}) \Rightarrow (eval(\texttt{null}, Env, Pow) = (Env, Pow))$

- *Wait Statement*

  $undef(\texttt{wait } \ldots) \Rightarrow (eval(\texttt{wait } \ldots, Env, Pow) = (Env, Pow))$

Until now we have introduced fault modes without using expression replacements. The following fault mode behavior models use expression replacements which are computed using the algorithm REPLACEMENTS of page 97.

- *Signal Assignment*

  - S = "T`<=`E `after` TE"

    $target_{NT}(S) \Rightarrow (eval(S, Env, Pow) =$
    $\quad (Env, (Pow(NT))[(eval_E(E, Env), ActualTime + eval_E(TE, Env)]_{INERT}))$

    $expr_{NE}(S) \Rightarrow (eval(S, Env, Pow) =$
    $\quad (Env, (Pow(T))[(eval_E(NE, Env), ActualTime + eval_E(TE, Env)]_{INERT}))$

    $time_{NTE}(S) \Rightarrow (eval(S, Env, Pow) =$
    $\quad (Env, (Pow(T))[(eval_E(E, Env), ActualTime + eval_E(NTE, Env)]_{INERT}))$

  - S = "T`<= transport` E `after` TE"

    $target_{NT}(S) \Rightarrow (eval(S, Env, Pow) =$
    $\quad (Env, (Pow(NT))[(eval_E(E, Env), ActualTime + eval_E(TE, Env)]_{TRANS}))$

    $expr_{NT}(S) \Rightarrow (eval(S, Env, Pow) =$
    $\quad (Env, (Pow(NT))[(eval_E(NE, Env), ActualTime + eval_E(TE, Env)]_{TRANS}))$

    $time_{NTE}(S) \Rightarrow (eval(S, Env, Pow) =$
    $\quad (Env, (Pow(NT))[(eval_E(E, Env), ActualTime + eval_E(NTE, Env)]_{TRANS}))$

- *Variable Assignment* S = "T `:=` E "

$$target_{NT}(S) \Rightarrow (eval(S, Env, Pow) = \\ (Env[(NT, eval_E(E, Env))], Pow))$$

$$expr_{NE}(S) \Rightarrow (eval(S, Env, Pow) = \\ (Env[(V, eval_E(NE, Env))], Pow))$$

- *Conditionals* S = "`if` E `then` SB_1 `else` SB_2 `end if`"

$$cond_{NE}(S) \Rightarrow (eval_E(NE, Env) = true \rightarrow \\ eval(S, Env, Pow) = eval(SB_1, Env, Pow))$$

$$cond_{NE}(S) \Rightarrow (eval_E(NE, Env) = false \rightarrow \\ eval(S, Env, Pow) = eval(SB_2, Env, Pow)$$

- *Loops* S = "`while` E `loop` SB `end loop`"

$$cond_{NE}(S) \Rightarrow (eval_E(NE, Env) = true \rightarrow \\ eval(S, Env, Pow) = eval(S, Env', Pow')) \\ \text{where } (Env', Pow') = eval(SB, Env, Pow)$$

$$cond_{NE}(S) \Rightarrow (eval_E(NE, Env) = false \rightarrow \\ eval(S, Env, Pow) = (Env, Pow))$$

Observations are given by an oracle or the user by specifying the environment and projected output values occuring after a statement has been executed. An observation contradicts the program (process) evaluation if the given environment (or POW) is different to the one computed. To specify an observation exactly not only the statement but also the environment before executing the statement (called input environment) must be given. If this is not the case the expected (measured) environment (POW) must hold for all input environments. Note that it is not necessary that the values for all variables and signals must be given as input. A partial environment is sufficient.

We assume that an observation is given by $obs(St, Env_{IN}, Env_{OUT}, Pow_{OUT})$.

If the observation is not contradictory the following rule must hold:

$$(eval(St, Env, Pow) = (Env', Pow') \wedge \forall X : Env_{IN}(X) \neq \epsilon \rightarrow Env_{IN}(X) = Env(X)) \Rightarrow \\ ((\forall Y : Env_{OUT}(Y) \neq \epsilon \rightarrow Env_{OUT}(Y) = Env'(Y)) \wedge \\ (\forall Z : Pow_{OUT}(Z) \neq \epsilon \rightarrow Pow_{OUT}(Z) = Pow(Z)))$$

Therefore for every observation this rule is part of the system description.

It now remains to prove that the system description is correct with respect to the VHDL semantics.

Theorem 5.2.1 (Correctness of SD_ER) *Let P be a process, (Env, Pow) the input environment and the input projected output waveform (POW), and Env_O, Pow_O the expected output environment and POW. The system description is correct wrt. the program semantics because it delivers the same results if all components are assumed to behave correctly.*

$eval(P, Env, Pow) = (Env_O, Pow_O) \Rightarrow (SD \cup \{eval(P, Env, Pow) = (Env_O, Pow_O)\} \cup \bigcup_{C \in COMPS}\{ok(C)\} \not\models \perp$

The proof is simple and follows directly from the definition of **SD_ER** by comparing it rule by rule with the given semantics definition.

5.3 Using SD_ER: A short Example

After describing how to create the system description **SD_ER** we show how it can be used to find bugs within a short program. To do this, we use the entity `MEM` and architecture `BEHAV_2` introduced in example 1.1 (see page 3) and assume further that `encrypt`, `decrypt` and `decrypt_new` are elements of the function environment used.

Possible replacements for `decrypt(mem)` with a difference of 1 are `mem`, `not(mem)`, `decrypt_new(mem)`, leading to the wrong expression fault modes $expr_{mem}$, $expr_{not(mem)}$, $expr_{decrypt\_new(mem)}$ for the statement `data <= decrypt(mem)`.

Using the above system description and fault modes we obtain the single diagnosis $\{expr_{decrypt\_new(mem)}($`data <= decrypt(mem)`$)\}$. Of course other diagnoses are computed, too. This include the *undef* diagnosis for both conditional statements. However by using fault probabilities we can focus on the most relevant diagnoses. In this case the most relevant diagnosis is a diagnosis explaining the faulty behavior by showing how to correct it.

What we have seen is that the new system description **SD_ER** does really improve the diagnosis process by showing also how to correct a detected bug. In general the new system description does not guarantee to deliver less diagnosis candidates than the other system descriptions. To avoid such problems we can use the system description **SD_SEQ** to compute an initial component focus for **SD_ER**. As experiments has shown this will not only decrease the number of diagnoses but also increases time performance.

5.4 Complexity considerations

Since 1.2.2 has shown that the time complexity of the DIAGNOSE algorithm is worst case exponential and this is also true for the REPLACEMENT algorithm (see 5.1.1), we have to take care to avoid computational explosion. Searching only for single diagnosis has a worst case complexity of $o(|MODES| \cdots |COMPS|)$ where $|MODES|$ itself depends on $o(|FUNC|^D)$ where $FUNC$ is the function environment and D the expression difference.

Because of the expected time complexity it is necessary to restrict either the number of different fault modes and the number of components before doing diagnosis. Without using additional information this can be done by:

1. doing iterative diagnosis. First start with the expression replacements with a difference of one. Increase the difference until an acceptable diagnosis has been detected.
2. using the system description **SD_SEQ** to create a focus component set which is then used as diagnosis component set for **SD_ER**

The first approach should increase overall performance while guaranteeing a solution. The order of time complexity is the same as for starting diagnosis with a fixed difference since iterative algorithms do not change the order of time complexity. In the second approach the number of diagnosis components will be decreased on average. Hence performance will be increased.

The improvement of time performance can be further increased by using semantic information while creating expression replacements. In principle we use the same mechanism as for diagnosis of sequential programs. We associate every expression and subexpression with a diagnosis component and use an adapted version of SD_ER (SD_EXP) to find correct replacements.

First we define the conversion function transforming expressions to diagnosis systems. $conv_{exp}$ is defined as follows:

- Expr = "E_1 `op` E_2"

  $conv_{EXP}(Expr, Env) = (out_{Expr}, Comp_1 \cup Comp_2 \cup \{Expr\}, SD_1 \cup SD_2 \cup SD)$

  with

  $(Top_1, Comp_1, SD_1) = conv_{EXP}(E_1, Env)$

  $(Top_2, Comp_2, SD_2) = conv_{EXP}(E_2, Env)$

  $$SD = \{ok(Expr) \Rightarrow out_{Expr} = op(Top_1, Top_2),\\ \quad expr_{Repl}(Expr) \Rightarrow out_{Expr} = eval_E(Repl, Env)\}$$

- Expr = "`si` E_1"

  $conv_{EXP}(Expr) = (out_{Expr}, Comp_1 \cup \{Expr\}, SD_1 \cup SD)$

  with

  $(Top_1, Comp_1, SD_1) = conv_{EXP}(E_1, Env)$

  $$SD = \{ok(Expr) \Rightarrow out_{Expr} = op(Top_1),\\ \quad expr_{Repl}(Expr) \Rightarrow out_{Expr} = eval_E(Repl, Env)\}$$

- Expr = "`(` E `)`"

  $conv_{EXP}(Expr) = conv_{EXP}(E, Env)$

- Expr = "I`(` $E_1, \ldots, E_n$ `)`" where I denotes a user-defined function.

  In this case we ignore the underlying function definition.

  $conv_{EXP}(Expr) = (out_{Expr}, \{Expr\}, SD)$

  with

$$SD = \{ok(Expr) \Rightarrow out_{Expr} = eval_E(Expr, Env),$$
$$wexpr_{Repl}(Expr) \Rightarrow out_{Expr} = eval_E(Repl, Env)\}$$

Note that using the information stored within a function body can improve the result of the diagnosis process. The current model does not allow back propagation and therefore may deliver too many diagnosis candidates. However, user-defined functions are not very often used in todays VHDL programs.

- Expr = "$I($ E $)$" where I denotes a variable or signal storing an array.

  $conv_{EXP}(Expr) = (out_{Expr}, Comps_1 \cup Comps_I, SD_1 \cup SD_I \cup SD)$

  $(Top_1, Comps_1, SD_1) = conv_{EXP}(E, Env)$

  $(Top_I, Comps_I, SD_I) = conv_{EXP}(I, Env)$

  $$SD = \{ok(Expr) \Rightarrow out_{Expr} = Top_I(Top_E),$$
  $$expr_{Repl}(Expr) \Rightarrow out_{Expr} = eval_E(Repl, Env)\}$$

- Expr = "I" where $I \in CONST$

  $conv_{EXP}(Expr) = (out_{Expr}, \{Expr\}, SD)$

  with

  $SD = \{ok(Expr) \Rightarrow out_{Expr} = I, expr_{Repl}(Expr) \Rightarrow out_{Expr} = eval_E(Repl, Env)\})$

- Expr = "V" where $V \in VARS \cup SIG$

  $conv_{EXP}(Expr) = (out_{Expr}, \{Expr\}, SD)$

  with

  $$SD = \{ok(Expr) \Rightarrow out_{Expr} = Env(V),$$
  $$expr_{Repl}(Expr) \Rightarrow out_{Expr} = eval_E(Repl, Env)\})$$

The idea of searching for replacements using the given semantics is as follows. First compute all diagnoses using only the ok and $undef$ mode. Use the resulting diagnoses as component focus for diagnosing the expression using ok, $undef$ and $expr$. In the second step, the replacements are restricted. Functions (operations) can only be replaced by other functions (operations) and variables, signals, and constants can be replaced by expression of a given size. The resulting replacements explaining the faulty behavior are then replacements for expression faults of sequential statements.

REPLACE_SEM(E,Val,Env,FEnv)

1. Convert the expression into SD and $COMP$ using the $conv_{EXP}$ conversion function, i.e., $(Top, SD, COMP) = conv_{EXP}(E)$. The observations are given by Env and $\{Top = Val\}$.

2. Let $DIAGS$ be the set of diagnoses computed using SD, $COMP$, OBS by restricting the fault modes to ok and $undef$.

3. Let $IMPR$ be the set of diagnoses computed using SD, $COMP$, OBS and all modes, where the replacements are computed as follows:
   - Forall $undef(E_1 \texttt{ op } E_2) \in DIAGS$ and all functions (operations) F with the same input and output types than `op` define the mode $expr_{E_1 \texttt{ F } E_2}$ for component $(E_1 \texttt{ op } E_2)$.
   - Forall $undef(\texttt{op } E_1) \in DIAGS$ and all functions (operations) F with the same input and output types than `op` define the mode $expr_{\texttt{F } E_1}$ for component $(\texttt{op } E_1)$.
   - Forall $undef(X) \in DIAGS$ where X denotes a variable, signal or constant use the REPLACEMENT algorithm to create possible fault modes.
4. The resulting $expr_{Repl}(Expr)$ diagnoses are returned as modes for the expression $Expr$.

The above algorithm for finding replacements after using a less concise system description has been implemented in Smalltalk and tested using various examples. We have found that using only replacements without focusing on relevant subexpressions is allways more time consuming than the presented algorithm (except in the cases where the focus set is equal to the diagnosis component set).

Example 12: Consider the following expression consisting out of 29 subexpression indicated by indices beside each operator and variable:

$$\begin{array}{l}
\texttt{(((Bn}_5\texttt{ and}_4\texttt{ An}_6\texttt{) or}_3\texttt{ Cn}_7\texttt{) or}_2 \\
\texttt{((not}_{10}\texttt{(Bn}_{11}\texttt{) and}_9\texttt{ An}_{12}\texttt{) and}_8\texttt{ not}_{13}\texttt{(Cn}_{14}\texttt{))) or}_1 \\
\texttt{(((not}_{18}\texttt{(Bn}_{19}\texttt{) and}_{17}\texttt{ not}_{20}\texttt{(An}_{21}\texttt{)) and}_{16}\texttt{ Cn}_{22}\texttt{) or}_{15} \\
\texttt{((Bn}_{25}\texttt{ and}_{24}\texttt{ not}_{26}\texttt{(An}_{27}\texttt{)) and}_{23}\texttt{ not}_{28}\texttt{(Cn}_{29}\texttt{)))}
\end{array}$$

and assume the environment $Env(An) =' 0'$, $Env(Bn) =' 1'$, $Env(Cn) =' 1'$ and expected result '0'. The set of all single diagnosis is

$\{\{Cn_7\}, \{or_1\}, \{or_2\}, \{or_3\}\}$

when using only ok and $undef$. Replacements computed are

$\{(Cn_7, Bn), (Cn_7, not(Cn)), (or_1, and), (or_2, and), (or_3, and)\}$.

The following table gives the amount of time and the number of diagnoses computed for every diagnosis step. In the last step, we only search for replacements with a difference of 1.

Diagnosis	Time [s]	Number of Diags
Diag. ok,$undef$	1.3	4
Diag. ok,$undef$, $expr$	7.8	5
Focused Diag.	1	5

The focused algorithm REPLACE_SEM improves the search for replacements by factor of about 8 in this example. □

Using diagnoses to find expression replacements has another advantage. Since the structure of the resulting diagnosis system is a tree, very fast diagnosis algorithms can be used. See [FD95] for a diagnosis algorithm using constraints.

Chapter 6

Problems, Restrictions and Extensions

6. Die allgemeine Form der Wahrheitsfunktion ist: $[\bar{p}, \bar{\xi}, N(\bar{\xi})]$.
Dies ist die allgemeine Form des Satzes.

Ludwig Wittgenstein, 1921
Tractatus logico-philosophicus
Logisch-philosophische Abhandlung

6.1 Diagnosing Missing Statements

The presented approach of using model-based diagnosis techniques for debugging of imperative programming languages has the disadvantage of not being able to deal with missing functionality in general, e.g., a missing assignment statement. We introduce the problem by using example programs with missing functionality.

Example 13:

The VHDL program

```
entity mf is
end mf;
architecture ex1 of mf is
     signal S1,S2:  std_logic := '0';
begin
     p1:  process
     begin
          -- S1 <= not S2 after 20 ns; Missing Statement
```

```
        wait until false;
    end process p1;
end ex1;
```

shows an architecture with a process not changing the value of a signal. Assume for example that the result of the above program should be a change of signal `S1` to `'1'` at 20ns. Because of a wrong value at 20ns (`'0'` instead of `'1'`) the diagnosis algorithm should deliver a subset of all diagnosis components as result. Unfortunatelly this is not the case for the three introduced system descriptions in general. The diagnosis algorithm using the system descriptions **SD_FSD** and **SD_SEQ** will answer $\{\}$ as diagnosis. The system description **SD_ER** may deliver diagnoses depending on the given similarity value between program and correct program. □

Example 14: The second VHDL program

```
architecture ex2 of mf is
    signal S1,S2:  std_logic := '0';
begin
    p1:  process
        variable V: std_logic := '1';
    begin
        -- V := not S2 ; Missing Statement
        S1 <= not V;
        wait until false;
    end process p1;
end ex2;
```

shows the case where a missing statement interacts with existing one. All three system description will provide a result in this case. **SD_FSD** returns $\{p1\}$ as result. An extended version using functional dependencies for diagnosis of sequential statements will additionally return $\{S1\texttt{<=}notV;\}$ as result. **SD_SEQ** will deliver $\{undef(S1\texttt{<=}notV;)\}$ as diagnosis, and **SD_ER** $\{wrongE_V(S1\texttt{<=}notV;)\}$. In this case the computed diagnoses are not the correct ones. However, they do provide information about where to look at. In any case, either one of the diagnoses is correct, or there must be a missing statement which should be used before a statement which is part of a diagnosis. □

The problem introduced can be solved either by using meta rules for diagnosis result interpretation or (explicitly) by introducing diagnosis components without functionality during conversion. The first solution can be expressed by the following two meta rules.

1. *If no diagnoses can be found there must be at least one missing assignment statement with the signal causing the discrepancy as target. The location of this statement is not known at all.*
2. *For all diagnoses, the possibility remains that a missing statement may cause the misbehavior. The statements can be missing everywhere in the program.*

Missing statements can be explicitly represented by introducing statements of the form S<=S for signals and $V := V$ for variables at the beginning of a process before conversion. The introduction of additional statements makes it possible to refer to a statement in case of missing functionality and known misbehavior. It does not help to fix the real fault since it is not necessarily the first statement that is wrong.

6.2 Diagnosing Type Faults

The previously described models do not take account of type faults. They assume that the program to be diagnosed has no faults regarding wrong types. However we argue that the introduced approach is general and easy expandable allowing to diagnose type faults as well needing only a few additional modeling. Consider for example the VHDL program

```
architecture BEH of entity TEST is
     signal S,A: std_logic_vector := "1111";
     signal B: integer := 0;
begin
     P1:process
     begin
          S <= A and B;
     end process P1;
end BEH;
```

where every compiler implementing the VHDL standard should indicate a type error. In our case, after convertion a type error does not need to occur depending on the underlying functional representation. If, for example, the *and* is defined using first order logic such as

$and('1','1','1'), X \neq' 1' \rightarrow and(X,Y,'0'), Y \neq' 1' \rightarrow and(X,Y,'0'),$

then no type error occur. If we use a partially defined function *and* it must be specified what should happened calling it with wrong values. Because **SD_SEQ** and **SD_ER** use functions with an assumed underlying behavior we show for this case how type errors can be detected and integrated into the existing debugging model.

To do this, we only need to redefine the behavior of the functions. We assume that every function used returns a distinct value *type* if a type error occurs. In addition, we insert the rule

$eval_E(E, Env) = undef \Rightarrow \perp$

into **SD_ER** and

$x = undef \Rightarrow \perp$

into **SD_SEQ** where x denotes a connection between diagnosis components.

This concept can be extended to signal (variable) assignments and other statements.

A contradiction caused by $eval_E$ can only occur if statements are not assumed to be incorrect. Therefore the creation of hitting sets is possible. Hence diagnosis of type faults is possible.

6.3 Combining Verification and Debugging Techniques

As stated in chapter 1, verification is different from diagnosis. While verification solves the problem of saying that a program behaves as expected, diagnosis gives an explanation why a program does not behave as expected. Using this view, the combination of both techniques seams to be a valuable step in the right direction.

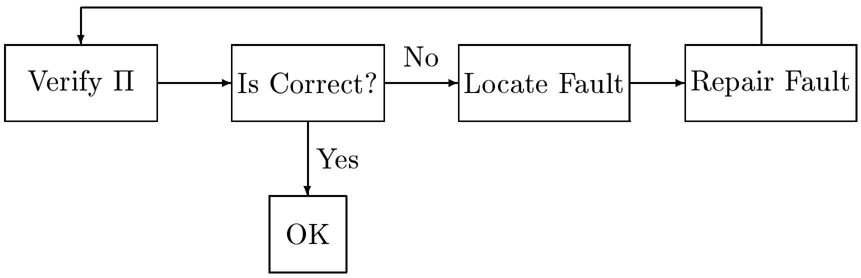
The integration of verification and program debugging can be done by

1. Using the verification information, or by
2. Using test cases verification tools usually provide to show when a program behaves incorrect.

Using verification directly for program debugging as we know has never been done before. However, it may be possible to use the underlying verification system model as model for diagnosis. On the other hand, symbolic diagnosis using symbolic equations rather than values may also be possible. A detailed answer of how those ideas can be used is still missing.

The information which a verification tool gives to the user after detecting that a program does not meet it's specification can be directly used for diagnosis because a contradictionary example is given back as verification result. This example usually contains input and (expected) output values.

The following figure shows how verification of a program Π interacts with automatic program debugging.



The tasks *Locate Fault* and *Repair Fault* can be summarized to program diagnosis as described in this thesis.

Chapter 7

Conclusion

7. Wovon man nicht sprechen kann, darüber muss man schweigen.

Ludwig Wittgenstein, 1921
Tractatus logico-philosophicus
Logisch-philosophische Abhandlung

This thesis presents three possible approaches of how to debug concurrent and sequential programs. The underlying ideas are examplified by programs written in the widely used hardware description language VHDL. All approaches have in common that the programs to be diagnosed are converted into a diagnosis system consisting of components and a system description. The system description contains the behavior of the components and their connections. Based on the converted programs, standard model-based diagnosis techniques including a diagnosis algorithm and a measurement selection algorithm can be used to find a bug within a program. Using repair, in our case replacements of statements and expressions to improve diagnosis capability is also not new, but has never been used in this domain.

The introduced system descriptions allow to find functional bugs but do not deal with bugs related to temporal faults. Although missing expressions and statements causing a bug are not focused on, ways of locating bugs in this case a presented. As future research it remains to improve the logic model by introducing concepts related to arrays, function calls, memory and others. The basic foundations have been presented.

Finally we introduce a comparison of the three models. We use the following parameters:

Capability describes the size of programs which can be diagnosed within an acceptable time. An acceptable time for debugging means about less than a minute.

Quality The diagnosis quality describes the result of the diagnosis process. Quality is related to the average number of diagnoses computed and the possibility of correcting a bug.

Measurement Selection Describes the capability of using measurement selection to improve diagnosis results.

Bug Correction answers wether the system description not only delivers the location of a bug, but also answers how to correct it.

The next table shows the result of comparison.

	SD_FSD	**SD_SEQ**	**SD_ER**
Capability	large	medium	medium/small
Quality	poor	good	very good
Funct. Bugs	yes	yes	yes
Bugs missing St.	no	no	yes
Meas. Sel.	restricted	yes	yes
Bug Correction	no	no but hints	yes

To improve the overall diagnosis performance using a specific system description it is a good idea of using less exact results as focus for the diagnosis algorithm using a better system description.

This thesis has described how (VHDL) programs can be converted to a system description used for model-based diagnosis. Three conversions have been considered. The first is a very abstract one only using the information about dependencies between signals and variables. The second conversion can be used for sequential statements and based on evaluation traces. It has been explained why standard model-based diagnosis techniques should be used for diagnosis of concurrent statements (with all the restricitions known from the hardware domain). The final conversion type uses expression replacements for diagnosis, allowing to repair faulty programs while diagnosing them. However, the underlying model of the third conversion has a time complexity which restricts usage only to very small programs. To avoid this problem focusing on most probable candidates by using less good system descriptions has been introduced. Although a way of using evaluation traces for computing possible replacements and practical implications has been part of this thesis.

Open problems not tackled by this thesis include system models for memory access and how to diagnose faults using pointers, generating missing statements, and a better more accurate solution for the detection of non-terminating programs. However, it has been shown that using model-based diagnosis for program debugging of sequential and concurrent programs is possible. Differences between program verification and program debugging and how to connect these two areas were presented, as well.

Bibliography

[AC91] Dean Allemang and B. Chandrasekaran. Functional representation and program debugging. In *Proc. IEEE Knowledge-Based Software Engineering Conference*, pages 136 – 143. IEEE, 1991.

[BGM94] Egon Börger, Uwe Glässer, and Wolfgang Müller. The semantic of behavioral vhdl'93 descriptions. In *Proceedings of the European Design Automation Conference with EURO-VHDL*, 1994.

[BH93] 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.

[BH95] Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, pages 31 – 41, 1995.

[Bha92] Jayaram Bhasker. *A VHDL Primer*. Prentice Hall Inc., 1992.

[Bon94] Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis*. PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Ontario, CANADA K1S 5B6, 1994.

[BP94] 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 K1S 5B6, 1994.

[CFD93] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1494–1499, Chambery, August 1993. Morgan Kaufmann Publishers, Inc.

[CWH94] Pi-Yu Chung, Yi-Min Wang, and Ibrahim N. Hajj. Logic design error diagnosis and correction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2:320–332, 1994.

[Dav84] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[DH88] Randall Davis and Walter Hamscher. Model-based reasoning: Troubleshooting. In *Exploring Artificial Intelligence*, chapter 8, pages 297–346. Morgan Kaufmann Publishers, Inc., 1988.

[dK91] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings AAAI*, pages 842–848, Anaheim, July 1991. Morgan Kaufmann Publishers, Inc.

[dKMR90] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnoses. In *Proceedings AAAI*, pages 324–330, Boston, August 1990. Morgan Kaufmann Publishers, Inc.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[DSH+82] R. Davis, H. Shrobe, W. Hamscher, K. Wieckert, M. Shirley, and S. Polit. Diagnosis based on structure and function. In *Proceedings AAAI*, pages 137–142, Pittsburgh, August 1982.

[dYG95] Byung do Yoon and Oscar N. Garcia. A cognitive framework of debugging. In *SEKE*, 1995.

[FD95] Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *Proc. IJCAI*, pages 1742 – 1748, 1995.

[Fer87] Gerard Ferrand. Error Diagnosis in Logic Programming, An Adaption of E.Y Shapiro's Method. *Journal of Logic Programming*, 1987:177–198, 1987.

[FSW95] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proceedings on the Sixth International Workshop on Principles of Diagnosis*, Goslar, October 1995.

[FSW96] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Budapest, August 1996.

[GC89] Ashok Goel and B. Chandrasekaran. Functional representation of designs and redesign problem solving. In *Proc. IJCAI*, pages 1388–1394, Detroit, August 1989. Morgan Kaufmann Publishers, Inc.

[GSW89] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[Ham88] Walter C. Hamscher. *Model-Based Troubleshooting of Digital Systems.* PhD thesis, MIT AI Lab, August 1988. Also appeared as AI-TR 1074.

[Ham91] Walter C. Hamscher. Modeling digital circuits for troubleshooting. *Artificial Intelligence*, 51(1-3):223–271, October 1991.

[HD84] Walter C. Hamscher and Randall Davis. Diagnosing circuits with state - an inherently underconstrained problem. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 276–282. Morgan Kaufmann Publishers, Inc., 1984.

[IC92] Y. Iwasaki and B. Chandrasekaran. Design Verification through Function- and Behavior-Oriented Representation. In *Artificial Intelligence in Design*, pages 597–616. Kluwer Academic Publishers, 1992.

[Jac95] Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[Joh86] W. Lewis Johnson. *Intention-Based Diagnosis of Novice Programming Errors.* Pitman Publishing, 1986.

[KB95] Carlos Delgado Kloos and Peter T. Breuer, editors. *Formal Semantics for VHDL* . Kluwer Academic Publishers, 1995.

[Liv94] Beat Liver. Modeling software systems for diagnosis. In *Proceedings on the Fifth International Workshop on Principles of Diagnosis*, pages 179–184, New Paltz, NY, October 1994.

[Llo87] J. W. Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5:133–154, 1987.

[MNPH90] R. Marcotte, M. Neiberg, R. Piazza, and L. Holtzblatt. Model-Based diagnostic reasoning using VHDL. Technical Report MTR11049, MITRE Corporation, December 1990.

[Mur88] William R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems.* Pitman Publishing, 1988.

[Nav93] Zainalabedin Navabi. *VHDL Analysis and Modeling of Digital Systems.* McGraw-Hill, 1993.

[Paw96] Thomas Pawlin. Implementierung eines C-Uebersetzers und Anwendung der modellbasierten Diagnose zum Software-Debugging von C-Programmen, 1996.

[RdKSS91] Olivier Raiman, Johan de Kleer, Vijay Saraswat, and Mark Shirley. Characterizing non-intermittent faults. In *Proceedings AAAI*, pages 849–854, Anaheim, July 1991. Morgan Kaufmann Publishers, Inc.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Sha83] Ehud Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, Massachusetts, 1983.

[SSGP92] E. Stroulia, M. Shankar, A. Goel, and L. Penberthy. A model-based approach to blame-assignment in design. In *Artificial Intelligence in Design*, pages 519–537, Pittsburgh, 1992.

[SW95] Markus Stumptner and Franz Wotawa. Modeling VHDL Programs for Diagnosis with Linear Computational Complexity. Technical Report DBAI-MBD-TR-95-03, Technische Universität Wien, June 1995.

[SW96a] Markus Stumptner and Franz Wotawa. Model-based diagnosis of hardware description languages. In *Proc. Computational Engineering in Systems Applications CESA'96 IMACS Multiconference, Invited Session on Model-Based Systems*, Lille, 1996. To appear.

[SW96b] Markus Stumptner and Franz Wotawa. Model-based program debugging and repair. In *Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Fukuoka, 1996. To appear. Available as technical report DBAI-MBD-TR-96-06.

[SW96c] Markus Stumptner and Franz Wotawa. A model-based tool for finding faults in hardware designs. In *Artificial Intelligence in Design*, Stanford, 1996. To appear. Available as technical report DBAI-MBD-TR-96-05.

[Ung91] Roland Unger. Ein Modellbasiertes Diagnosesystem - Erweiterung und Implementierung des Ansatzes nach Reiter, 1991.

[VHD88] IEEE Standard VHDL Language Reference Manual LRM Std 1076-1987, 1988.

[VKvB86] Marc Vilain, Henry Kautz, and Peter van Beek. Constraint propagation algorithms for temporal reasoning: A revised report. In *Proceedings AAAI*, pages 377–382, 1986.

[Wil86] R. Williams. Doing time: Putting qualitative reasoning on firmer ground. In *Proceedings AAAI*, pages 105–112, 1986.

[WL69] R. J. Waldinger and R. C. T. Lee. PROW: A step toward automatic program writing. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Washington, DC, May 1969.

Appendix A

RVHDL Syntax

To define the syntax we use the standard BNF. Syntax rules defining a nonterminal A are defined by A `::=` B_1 `|` ... `|` B_n where each B_i denotes a sequence of nonterminals and terminals or ϵ. ϵ is used to denote a string of length zero. Sometimes we use braces [] to denote optional parameters. Terminals are written in typed text style whereas nonterminals have no special text style.

The syntax of **RVHDL** is defined as follows:

```
vhdl_program ::=
    entity_declaration
    | architecture_declaration
entity_declaration ::=
    entity id is
        [ port (
            interface_declarations ) ; ]
    end id ;
interface_declarations ::=
    interface_declaration int_decl_rest
int_decl_rest ::=
    ; interface_declaration int_decl_rest
    | ε
interface_declaration ::=
    signal id_list : [ mode ] type_id [ := expr ]
    variable id_list : [ mode ] type_id [ := expr ]

mode ::=
    in | out | inout
architecture_declaration ::=
    architecture id of id is
        architecture_declarations
    begin
```

```
        concurrent_statements
    end  id ;
concurrent_statements ::=
    process_statement concurrent_statements
    | ϵ
process_statement ::=
    id :   process
        process_declarations
    begin
        sequential_statements
        wait_statement ;
    end process id ;
sequential_statements ::=
    sequential_statement ;  sequential_statements
    | ϵ
sequential_statement ::=
    signal_assignment
    | variable_assignment
    | if_statement
    | loop_statement
    | null_statement
signal_assignment ::=
    id <= [ transport ] expr after  expr
variable_assignment ::=
    id :=  expr
if_statement ::=
    if  expr then
        sequential_statements
    else
        sequential_statements
    end if
loop_statement ::=
    while expr
    loop
        sequential_statements
    end loop
null_statements ::=
    null
wait_statement ::=
    wait  wait_statement_rest
wait_statement_rest ::=
    on  id_list
    | for  expr
    | until expr
architecture_declarations ::=
```

```
      signal_declaration ; architecture_declarations
      function_declaration ; architecture_declarations
      | ε
signal_declaration ::=
      signal id_list : type_id [ := expr ]
function_declaration ::=
      function id [ ( interface_declarations ) ] is
            process_declarations
      begin
            sequential_statements
            return expr ;
      end id ;
process_declarations ::=
      variable_declaration ; process_declarations
      function_declaration ; process_declarations
      | ε
variable_declaration ::=
      variable id_list : type_id [ := expr ]
id_list ::=
      id id_list_e
id_list_e ::=
      , id id_list_e
      | ε
expr ::=
      relation [ log_op relation ]
log_op ::=
      and | or | xor | nand | nor
relation ::=
      simpl_expr [ rel_op simpl_expr ]
rel_op ::=
      = | /= | < | <= | > | >=
simp_expr ::=
      [ sign ] term [ add_op term ]
sign ::=
      + | -
add_op ::=
      + | - | &
term ::=
      factor [ mult_op factor ]
mult_op ::=
      * | / | mod | rem
factor ::=
      primary [ ** primary ]
      | abs primary
```

| `not` primary

primary ::=

id primary_rest

literal

(expr)

primary_rest ::=

(expr_list)

| ϵ

expr_list ::=

expr expr_list_rest

expr_list_rest ::=

, expr expr_list_rest

| ϵ

Acknowledgment

First of all I wish to thank my parents for support and enabeling me to visit university and to complete my studies. My special thanks also go to my wife Michaela for her love and understanding.

It's a pleasure to acknowledge the support of the head of our department Georg Gottlob who makes research in the area of program debugging and model-based diagnosis possible. I also wish to thank Gerhard Friedrich, who introduced me into the concepts of model-based reasoning. Special thanks goes to Markus Stumptner for very fruitful discussions and all of his valueable comments on previous versions of this thesis.

Finally, I wish to thank the rest of my family, and all of my friends for their help and support.

Curriculum Vitae

Persönliche Daten

Name	DI Franz Wotawa
Geboren am	16.12.1968
Staatsbürgerschaft	Österreich
Religion	röm.kath.
Stand	verheiratet (Michaela geb. Forstner), 1 Kind (Franz Wolfgang)
Eltern	Eveline (geb. Deutsch) und Franz Wotawa
Interessen, Hobbys	AI, Logik, Kunst (Bilder, Filme), Photographie, Philosophie (Wittgenstein), Schwimmen, Skifahren

Ausbildung

1975 - 79	Volksschule in Wien Hernals/Halirschgasse
1979 - 83	Bundesrealgymnasium Wien Geblergasse
1983 - 88	Technologisches Gewerbemuseum (TGM) Wien Brigittenau Höhere Technische Bundeslehr- und Versuchsanstalt Fachrichtung Elektrische Nachrichtentechnik und Elektronik
1988 -94	Studium der Informatik an der Technischen Universität Wien mit Wahlfach Artificial Intelligence. Diplomarbeit "Analyse und Algorithmen von fokusierten Assumption Based Truth Maintenance Systemen".
Seit 1994	Doktorratsstudium Informatik an der Technischen Universität Wien

Tätigkeiten

1984 - 1988	Ferialpraktika bei der Firma Siemens
1988 - 1993	Zweimalige Ferialpraxis bei IBM Österreich (Entwurf und Programmierung von Applikationen, Trainer für Standardsoftware (Pagemaker, Corel Draw) im Rahmen eines von IBM organisierten Computercamps. Freiberuflicher Mitarbeiter der ÖMV, Abteilung MI-AI (Entwurf und Implementierung von Informationssystemen für den internen Gebrauch. Entwurf von Datenbanken und Benutzerschnittstellen).
1994 - 1995	Wissenschaftlicher Assistent am Institut für Informationssysteme, Abteilung Datenbanken und Expertensysteme, im Rahmen des DDV-Projekts der Firma Siemens. Erarbeitung der Grundlagen Modellbasierte Diagnose auf Programmiersprachen anwendbar zu machen. Objektorientierter Entwurf und Implementierung eines Diagnosesystems für die Hardware-Entwurfssprache VHDL.
Seit 1.4.1995	Universitätsassistent am Institut für Informationssysteme, Abteilung für Datenbanken und Expertensysteme. Fachgebiete: Diagnose von Software und Hardware, Modellierung für die AI, Objektorientierter Entwurf und Implementierung