Adaptable rule checking tools for HDL

Examensarbete utfört i Elektroniksystem vid Tekniska högskolan i Linköping
av
Mikael Lord
LiTH-ISY-EX--09/4080--SE
Linköping 2009
Adaptable rule checking tools for HDL
Adaptable rule checking tools for HDL

Today's electronics in aviation (avionics) are more complex than ever before. With higher requirements on safety and reliability and with new SoC (System on Chip) technology, the validation and verification of designs meet new challenges.

In commercial and military aircraft there are many safety-critical systems that need to be reliable. The consequences of a failure of a safety-critical system on-board a civil or military aircraft are immeasurably more serious than a glitch or a bit-flip in a consumer appliance or Internet service delivery. If possible hazards are found early in the design process, a lot of work can be saved later on. Certain structures in the code are prone to produce glitchy logic and timing problems and should be avoided.

This thesis will strengthen Saab Avitronics knowledge of adaptable rule checking tools for HDL, with a market analysis of the tools available. Moreover will it evaluate two of the most suitable tools and finally it will describe some of the design issues that exist when coding safety-critical systems.

Finally it is concluded that the introduction of static rule checking tools will help the validator to find dangerous constructs in the code. However, it will not be possible to fully automate rule checking for safety-critical systems, because of the high requirements on reliability.

**Keywords**
Lint, rule checking, safety-critical, VHDL

---

**Title**
Adaptable rule checking tools for HDL

**Author**
Mikael Lord
Abstract

Today’s electronics in aviation (avionics) are more complex than ever before. With higher requirements on safety and reliability and with new SoC (System on Chip) technology, the validation and verification of designs meet new challenges.

In commercial and military aircraft there are many safety-critical systems that need to be reliable. The consequences of a failure of a safety-critical system onboard a civil or military aircraft are immeasurably more serious than a glitch or a bit-flip in a consumer appliance or Internet service delivery. If possible hazards are found early in the design process, a lot of work can be saved later on. Certain structures in the code are prone to produce glitchy logic and timing problems and should be avoided.

This thesis will strengthen Saab Avitronics knowledge of adaptable rule checking tools for HDL, with a market analysis of the tools available. Moreover will it evaluate two of the most suitable tools and finally it will describe some of the design issues that exist when coding safety-critical systems.

Finally it is concluded that the introduction of static rule checking tools will help the validator to find dangerous constructs in the code. However, it will not be possible to fully automate rule checking for safety-critical systems, because of the high requirements on reliability.
Acknowledgments

This thesis is part of the graduation for Master of Science in Applied physics and Electrical engineering at Linköping University. The work is performed at Saab Avitronics in Jönköping during the autumn of 2007 and the early months of 2008.

I would like to thank Saab Avitronics for providing me with an office and computer and I also want to thank the following people.

PhD Håkan Forsberg at Saab Avitronics for helping guiding me and answering different questions and introducing me to the subject.

Bengt Jönsson at Saab Avitronics for providing me with aviation code.

Bo Alpteg at Saab Avitronics for helping me install programs.

Kent Palmkvist at ISY, Linköping University, Examiner.
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>CDC</td>
<td>Clock Domain Crossing</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma Separated Value</td>
</tr>
<tr>
<td>DFT</td>
<td>Design For Test</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GREP</td>
<td>General Regular Expression Print</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper Text Mark-up Language</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>MBU</td>
<td>Multiple Bit Upset</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>OpenMORE</td>
<td>Open Measure Of Reuse Excellence</td>
</tr>
<tr>
<td>OVA</td>
<td>OpenVera</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>PSL</td>
<td>Property Specification Language</td>
</tr>
<tr>
<td>RMM</td>
<td>Reuse Methodology Manual</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SDC</td>
<td>Synopsys Design Constraint</td>
</tr>
<tr>
<td>SEE</td>
<td>Single Event Effect</td>
</tr>
<tr>
<td>SEU</td>
<td>Single Event Upset</td>
</tr>
<tr>
<td>SoC</td>
<td>System on-a-Chip</td>
</tr>
<tr>
<td>STARC</td>
<td>Semiconductor Technology Research Center</td>
</tr>
<tr>
<td>SVA</td>
<td>SystemVerilog Assertions</td>
</tr>
<tr>
<td>Tcl</td>
<td>Tool Command Language</td>
</tr>
<tr>
<td>TMR</td>
<td>Triple Mode Redundancy</td>
</tr>
<tr>
<td>TSV</td>
<td>Tab Separated Value</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
<tr>
<td>VSIA</td>
<td>Virtual Socket Interface Alliance</td>
</tr>
</tbody>
</table>


Contents

1 Introduction 1

1.1 Problem .............................................. 2
1.2 Goal ................................................ 2
1.3 Method ............................................. 2
1.4 Limitations ......................................... 2
1.5 Structure ........................................... 2

2 Background 5

2.1 Rule checking tools ................................. 5
2.2 Safety-critical systems ............................. 6
2.3 Cosmic particle radiation ......................... 6

3 Principles of a rule checking tool for HDL 9

3.1 Structure ........................................... 9
3.2 User interface ...................................... 9
3.3 Rules .................................................
  3.3.1 Rule-sets ...................................... 11
  3.3.2 Specifying the rules ........................... 11
  3.3.3 Implementing the rules in the rule checker .... 11
  3.3.4 Networking .................................... 12
3.4 Setting-up the tool ................................ 12
  3.4.1 Exclusion ....................................... 12
3.5 Results ............................................. 12
3.6 Integration of a rule checker ...................... 12

4 Design issues in HDL for safety-critical systems 15

4.1 Timing problems ................................... 15
  4.1.1 Timing constraints ........................... 15
  4.1.2 Metastability .................................. 15
  4.1.3 Glitches ....................................... 16
4.2 Synchronous versus asynchronous designs ........ 16
4.3 Latches ............................................. 16
4.4 Flip-flops .......................................... 18
4.5 Clocks .............................................. 18
  4.5.1 Clock domain crossing ....................... 18

xii
## Contents

7.1.4 Result ................................................. 42
7.2 Specific applicability .................................. 45
    7.2.1 Rules ............................................. 45
    7.2.2 Implement specific rules ....................... 45
7.3 Reliability ............................................. 47
7.4 Running avionics codes .............................. 51
    7.4.1 Avionics code (Verified) ....................... 51
    7.4.2 Avionics code (not verified) .................. 52

8 Verification Navigator (VN-Check) ............... 53
    8.1 General applicability ............................ 53
        8.1.1 Running the tool ............................ 53
        8.1.2 Rules .......................................... 53
        8.1.3 Severity levels ............................... 54
        8.1.4 Result .......................................... 55
    8.2 Specific applicability ............................ 55
        8.2.1 Rules .......................................... 55
        8.2.2 Implement specific rules .................... 56
    8.3 Reliability .......................................... 59

9 Results .................................................. 61
    9.1 General applicability ............................. 61
    9.2 Specific applicability ............................. 62
    9.3 Reliability .......................................... 62
    9.4 Summary ............................................. 64
        9.4.1 Design Checker and its advantages and disadvantages. . . . 64
        9.4.2 VN-Check and its advantages and disadvantages. ......... 64

10 Conclusions ............................................ 65
    10.1 Motivation of achieved goals .................... 65
    10.2 Discussion about the result ..................... 65

11 Future work ........................................... 67

Bibliography .............................................. 69

A The rules that are implemented into Design Checker 71
B The rules that are implemented into VN-Check 75
C Testcode ................................................. 77
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Typical flow for rule checking tools.</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Different kinds of rules.</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Typical design flow for FPGA design.</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Setup and hold times</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>Inferred latch</td>
<td>17</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of gated clock</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>Example of gated clock and its timing diagram</td>
<td>19</td>
</tr>
<tr>
<td>4.5</td>
<td>Binary coding</td>
<td>20</td>
</tr>
<tr>
<td>4.6</td>
<td>Gray coding</td>
<td>21</td>
</tr>
<tr>
<td>4.7</td>
<td>One-Hot coding</td>
<td>22</td>
</tr>
<tr>
<td>4.8</td>
<td>Triple Modular Redundancy voting.</td>
<td>22</td>
</tr>
<tr>
<td>6.1</td>
<td>Block diagram of TEST_PROJECT.</td>
<td>38</td>
</tr>
<tr>
<td>7.1</td>
<td>Parameters for one of the RMM rules</td>
<td>42</td>
</tr>
<tr>
<td>7.2</td>
<td>Typical design quality report in Design Checker</td>
<td>44</td>
</tr>
<tr>
<td>7.3</td>
<td>Distribution of violations when testing avionics code (verified) in Design Checker</td>
<td>51</td>
</tr>
<tr>
<td>7.4</td>
<td>Distribution of violations when testing avionics code (not verified) in Design Checker</td>
<td>52</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Classifications of safety-critical systems in aviation. .................. 7

5.1 The tools that are included in the market analysis. ................... 26
5.2 Summary of the different tools and their capabilities. ................. 33

6.1 Testplan for evaluation of tools. ....................................... 36

7.1 General applicability with Design Checker. ............................ 43
7.2 Specific applicability with Design Checker. ............................ 45
7.3 TEST_CODE_entity.vhd violation report. .............................. 47
7.4 INTERNAL_COMP_entity.vhd violation report. ........................ 47
7.5 GATED_COMP_entity.vhd violation report. ............................ 48
7.6 CDC_COMP_entity.vhd violation report. .............................. 48
7.7 FSM_MOORE_COMP_entity.vhd violation report. ...................... 49
7.8 FSM_MEALY_COMP_entity violation report. ......................... 50

8.1 General applicability with VN-Check. ................................. 54
8.2 Specific applicability with VN-Check. ................................. 56
8.3 Result for TEST_PROJECT in VN-Check. .............................. 59

9.1 INTERNAL_COMP_entity.vhd ........................................... 62
9.2 GATED_COMP_entity.vhd .............................................. 62
9.3 FSM_MOORE_entity.vhd ................................................ 63
9.4 FSM_MEALY_entity.vhd ................................................. 63

A.1 Rules implemented into Design Checker ............................... 71
A.2 Rules implemented into Design Checker ............................... 72
A.3 Rules implemented into Design Checker ............................... 73

B.1 Rules implemented into VN-Check ..................................... 76
List of Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Forgetting the “else” in IF-statement.</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Signals that are clocked by two different clocks.</td>
<td>18</td>
</tr>
<tr>
<td>4.3 Binary coding</td>
<td>20</td>
</tr>
<tr>
<td>4.4 Gray coding</td>
<td>21</td>
</tr>
<tr>
<td>4.5 Gray coding</td>
<td>21</td>
</tr>
<tr>
<td>4.6 The “safe” attribute in VHDL</td>
<td>23</td>
</tr>
<tr>
<td>7.1 Graycode in Design Checker</td>
<td>46</td>
</tr>
<tr>
<td>8.1 HeaderComment in VN-Check</td>
<td>57</td>
</tr>
<tr>
<td>8.2 ConstantDeclarationComment in VN-Check</td>
<td>57</td>
</tr>
<tr>
<td>8.3 Naming in VN-Check</td>
<td>58</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Saab Avitronics develops and manufactures advanced electronics for military and civilian applications. It is a business unit within Saab AB which is a leading high technology company with business areas in defence, avionics and space. In Jönköping, Saab Avitronics develop qualified electronics, software and mechanics for aircraft, helicopters and other demanding applications. The majority of the projects are driven in an international environment and they delivers products to two of the world’s largest aircraft manufacturers, Airbus and Boeing.

The electronics systems in aviation (avionics) are constantly exposed to a harsh environment. Therefore it is crucial to design these systems with robustness in mind. Without this, an electronic failure in flight could immerse and cause an accident with major loss of life.

For the aviation industry, it has been impossible to use the latest technology that has not yet been fully tested, because of the high requirements for reliability. However, the technical development is pushing the aviation industry to reduce the size of the electronics and put more functionality in one chip. This has forced them to use FPGAs (Field Programmable Gate Arrays) and ASICs (Application Specific Integrated Circuits), which provide new challenges for reliability and verification.

The avionics is getting more and more complex, and the chances of making design errors increase with increasing complexity. However, new tools that help designers verify their designs are constantly introduced.

This thesis will look closer on rule checking tools for HDL (Hardware Description Language). Rule checking tools have been around for software languages like C and Java for a long time. In recent years, the market for rule checking tools has increased and a number of tools are now readily available for HDL as well. These tools will help to check the design for violations early in the design flow, violations that can cause problems later in the design. If these violations are found early, both time and cost can be reduced.
1.1 Problem

Here are the main problems:

- Today, the verification process consumes about 50-70% of the total FPGA design cycle, which is a lot. To save time and costs, it is desirable to reduce verification time without jeopardizing quality and safety.

- In the verification process, the code is reviewed manually by an experienced programmer. This is a very demanding task that is not fail-proof due to the human factor.

1.2 Goal

Here are the main goals of this thesis:

- Strengthen Saab Avitronics knowledge in adaptable rule checking tools for HDL.

- Suggest how they should implement their own rules and which tool(s) is/are the most suitable.

- Analyze some of the safety-critical problems that can arise when violating the rules.

1.3 Method

First a market analysis of the state-of-the-art tools is performed to get an overall view of the tools available and to strengthen Saab Avitronics knowledge in them. Then two of the tools are chosen based on preferences of how usable the tools will be for Saab. The tools are then evaluated and tested. Finally they are summarized for their advantages and disadvantages.

1.4 Limitations

In the market analysis the tools are selected based on how common they are. It is possible that some tools have been overseen, but in general the most important tools are in it. Because Saab prefers using Windows for HDL designs, only tools that supports Windows will be further evaluated and tested with code.

1.5 Structure

Starting in chapter 2, explaining what a rule checking tool is and how it works. Furthermore the chapter describes what a safety-critical system is and how the cosmic particle radiation affects the avionics. Chapter 3 explains in more detail
1.5 Structure

how a rule checking tool for HDL is working. Chapter 4 discusses design issues when designing safety-critical systems in aviation. Chapter 5 presents a market analysis of the available tools on the market. Chapter 6 explains the evaluation criteria's and also introduces a test code. In chapter 7, one of the selected tools is evaluated and in chapter 8 the second tool is evaluated. In chapter 9 there is a summary of the results and the tools are compared with their advantages and disadvantages. In chapter 10 there is a discussion about the result and finally in chapter 11 future work is discussed.
Chapter 2

Background

This chapter describes what a rule checking tool is and how it works. Furthermore it describes what a safety-critical system is and how the cosmic particle radiation affects the avionics.

2.1 Rule checking tools

A rule checking tool is a tool that checks the code against a set of rules. When the tool finds a violation against a rule, it gives some kind of violation message to the user.

Static analysis

A rule checking tool is a static analysis tool, which means that it analyzes the code without executing it, whereas a dynamic analysis tool is executing the code. Static analysis is a widely used term, but this thesis is focusing at rule checking tools.

Lint tools

Rule checking tools are also called Lint tools. The name Lint comes from the first code checking tool that was widely used, and was initially released for the UNIX environment by Bells Labs in the 70’s. The tool was able to check C and C++ programs for syntax and semantics errors. When checking the code for errors, this is commonly called Linting.

Different tools

According to [1] there are both academic and research based tools and also commercial. The academic tools are usually open source based and are inexpensive. The ones coming from research projects are usually not fully documented or supported. The commercial tools can be very expensive but are offering support.
The simplest tool for Linting is the pattern-based search tool. It can only search for text patterns in the code and flag if there is a match. E.g. of such tool is GREP (General Regular Expression Print) in UNIX. This tool uses regular expressions for finding patterns. These tools can help the programmer to find bugs to some extent, but can also flag for harmless things in the code.

There are also style checking tools that detect e.g. bad variable and method names, comment problems, indentation inconsistencies and other issues related to coding style. Another kind of tool is the semantic analysis tool that can detect e.g. data-type problems, uninitialized variables and unused methods. There are also tools for deep flow analysis that can capture faults like race conditions and deadlocks, pointer misuses etc.

There is a wide variety of tools available; from very simple to very advanced, and tools are available for most programming languages. Most tools that are available are for software languages like C, C++ and Java. However, this thesis will look closer on rule checking tools for HDL.

2.2 Safety-critical systems

A safety-critical system is a system whose failure or malfunction may result in:

- Death or serious injury to people, or
- Loss or severe damage to equipment or
- Environmental harm

For aviation, the classifications according to [2] are shown in table 2.1. The environment in aviation is very harsh and the electronics is e.g. exposed to large temperature differences, vibrations and cosmic particle radiation. To give an example, this thesis will look closer on the cosmic particle radiation.

2.3 Cosmic particle radiation

Cosmic particle radiation is high energy particles that travel from space down to earth through the atmosphere. They were first discovered by the Austrian physicist Victor Hess 1912 when he did a series of balloon flights and measured radiation levels. He found that the radiation is increasing with altitude.

The radiation at a typical cruising altitude for a long range airliner can be hundred times greater than on sea level where the vast majority of devices are designed to operate.

There is extensive research going on about cosmic particle radiation. NASA is one of the leading actors in this research area. They use this research when designing their spaceships and they have a lot of experience in the field.

According to [3], to see how the particle radiation affects the electronics, assume that a single particle is striking e.g one or several transistors in a memory cell. When one particle is striking one transistor/register this is called a SEU (Single
2.3 Cosmic particle radiation

<table>
<thead>
<tr>
<th>Level</th>
<th>Failure condition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Catastrophic</td>
<td>Failure condition which would result in multiple fatalities, usually with the loss of the airplane. Must be shown to be extremely improbable $p &lt; 10^{-9}$.</td>
</tr>
<tr>
<td>B</td>
<td>Hazardous</td>
<td>Failure condition which would lead to a large reduction in safety margins or functional capabilities, physical distress or excessive workload on the flight crew, or serious or fatal injury to a relatively small number of the occupants. Must be shown to be extremely improbable $p &lt; 10^{-7}$.</td>
</tr>
<tr>
<td>C</td>
<td>Major</td>
<td>Failure condition which would reduce the capabilities of the airplane or the ability of the crew to cope with adverse operating conditions. Must be shown to be improbable $p &lt; 10^{-5}$.</td>
</tr>
<tr>
<td>D</td>
<td>Minor</td>
<td>Failure condition which would not significantly reduce airplane safety.</td>
</tr>
<tr>
<td>E</td>
<td>No effect</td>
<td>Failure conditions that would have no effect on safety.</td>
</tr>
</tbody>
</table>

Table 2.1. Classifications of safety-critical systems in aviation.

Event Upset) or SEE (Single Event Effect). When one particle or several is striking multiple transistors/registers, this is called a MBU (Multiple Bit Upset) and this could be very dangerous.

When a particle is striking a memory cell, it can bit-flip the memory cell, causing the value of the cell to be changed from a zero to a one or vice-versa. This is also called a soft error and can result in failures in other electronics or strange behaviour. Memory circuits are the most vulnerable component in the electronic systems, because they contain large amounts of bits susceptible to SEU. If a particle strikes sensitive combinational logic, the voltage disturbance can propagate throughout the logic and affect the clock or asynchronous inputs of flip-flops, causing them register the wrong data.

There are also particles with higher energies that produce high currents in the device which can cause loss of device functionality. For these particles the electronics has to be shielded in some way.
Chapter 3

Principles of a rule checking tool for HDL

This chapter describes in more detail how a rule checking tool for HDL works.

3.1 Structure

According to the flowchart in figure 3.1, the HDL files are first loaded into the tool. Then the user defines a list of rules that is going to be used. The tool has access to a rule database, where all the rules are gathered. After running the tool on the code, a report file is generated. The report file shows which rules that are violated and on which lines the violations exist. If there are violations, the user needs to modify the code and run the tool again, otherwise the code has passed.

3.2 User interface

Most tools have a GUI (Graphical User Interface) and an API (Application Programming Interface) which is a programming environment that supports different scripting languages. Here the user can communicate with the tool in some extent and can write his/her own scripts. Using scripts can be useful when setting up the tool and/or choose which analyzes to perform.

3.3 Rules

The code checking tools use rules to check the code. The tools usually come with some already predefined rules which are parameterized. In some tools it is even possible to implement completely new rules.

There are different kinds of rules. Every tool has its own categories of rules, but the most common are (see figure 3.2):
• **Documentation rules:** Rules that make the code look more pretty and structured. Regulates e.g. how to indent the code and where there should be blank rows etc. Also things like comments and header layout can be found under this type of rules. The reason to have these kinds of rules is that it is easier to browse the code if it is well structured and therefore easier to find errors quickly. Also a common layout through a whole project can be achieved, despite that different people have written different parts of the code.

• **Naming rules:** Rules that regulate how names are chosen, e.g. processes, variables, signals, clocks, resets etc. Naming rules also includes different suffixes and prefixes for different types. E.g. can be `clock_n` for low clock edge, or `processname_proc` so it is easy to see what type it is and its use. Also lower- and uppercase are regulated here.

• **Coding rules:** Describes e.g. how to handle synchronous and asynchronous designs, clocks and resets, and FSMs (Finite State Machines).

• **Portability rules:** How to make the design possible to move to another platform (Make the design independent of platform).

• **Reusability rules:** How to make the code reusable for other designs.

• **Synthesizability rules:** How to make the design synthesizable.
3.3 Rules

The rules are usually divided into rule-sets. A common rule-set in most of the code checking tools is the RMM (Reuse Methodology Manual) [4] rule-set. The RMM is a collaborative project between Mentor Graphics and Synopsys and describes some good coding practices for reusing code in SoC (System-on-a-Chip) designs. The practices are based on experience from many design teams from all around the world.

3.3.2 Specifying the rules

Usually the rules that are going to be used in the tool are selected by a design team at the company. They know what kind of problems they use to have in their designs and what to look out for. Including RMM rules and best practices can be a good idea, to see what kind of violations exist in their code. Maybe the tool will find some things that the design team did not think of.

In the case of this thesis, Saab has already produced a document with all the rules that the programmer should comply with. The document [5] includes rules from the RMM and there are also some company specific rules for naming conventions and safety-critical systems. (This document is only for Saab’s internal usage and cannot be published in this thesis.)

3.3.3 Implementing the rules in the rule checker

It should be well documented which rules are implemented or not in the tool. Furthermore all deviations from the original rules must be well documented, so the
user is not thinking the rule is doing something, but instead it is doing something else.

3.3.4 Networking

The rule-sets can be available through a main server at the company, which ensures that all members of the design team are checking their code against the same set of rules. If a new rule is added or modified, the design team needs to know about this; otherwise new and conflicting rules can appear from one day to the next.

3.4 Setting-up the tool

Before the tool can start to analyze the code, the user needs to setup the tool and declare which projects/files to analyze, what rules to check against and also how the result should be presented.

3.4.1 Exclusion

Files can be excluded from being in the analysis. This can be very useful when e.g. a testbench is in the project. If only some content in a file needs to be excluded, exclusion pragmas can be used. The user sets a beginning pragma, which can be a comment in the code, and an ending pragma. The section between these pragmas will not be analyzed by the tool.

3.5 Results

The rule checking tools show where the violations are in the code and a report of the violations is presented. The rules usually have different severity levels if they are violated. To have some kind of quality assessment of the code, some tools include the OpenMORE (Open Measure Of Reuse Excellence) design scoring system which was devised by Synopsys and Mentor Graphics and is a points-scoring technique where, for each rule that is not violated, points are awarded. This means that the higher score, the better the design.

The results from the rule checking tool should be saved for future use. This gives statistics on which violations that were most common and it is also possible to observe how the quality of the code increases or decreases. This can help the design team to improve and know what to avoid.

3.6 Integration of a rule checker

According to [6], the integration of a rule checker can be done in the following way. First a specification is made of what the design should be like. Then the RTL (Register Transfer Level) code is generated. After this the code is checked in a HDL code checker. Because the HDL code checker is used early in the design flow, costly design iterations can be reduced as can be seen in figure 3.3.
Figure 3.3. Typical design flow for FPGA design.
Chapter 4

Design issues in HDL for safety-critical systems

In this chapter design issues when designing safety-critical systems in aviation are discussed.

4.1 Timing problems

In safety-critical systems, the timing is extremely important. There are two kinds of timing problems. First, the timing problems on a higher level in the software, which does not depend on failures in the hardware, but depends e.g. on collisions when cache memories are used. The second type of timing problem is on clock level, when the timing is not met in circuits. In this thesis timing problems in circuits are only of interest.

4.1.1 Timing constraints

Before an active clock edge (positive clock edge in figure 4.1) the data input has to be settled for at least the setup-time of the register. After an active clock edge the data input has to remain stable for at least the hold-time of the register. The setup-time and hold-time are illustrated in figure 4.1.

When the setup and hold time requirements are met there will be a valid output after $t_{CO}$ (clock to output delay).

4.1.2 Metastability

When the timing is not met, the flip-flop can take longer time than $t_{CO}$ to reach a valid level. This is called metastability and the output is hovering between 0 and 1. Metastability can cause the system to store wrong data or make delays in the system, which is why this should be avoided.
4.1.3 Glitches

If the timing constraints are not met, glitches may occur which can make certain components in the design to behave improperly. In ordinary consumer electronics, this is not a big deal but in safety-critical systems it is.

4.2 Synchronous versus asynchronous designs

In a synchronous design, events only happen on the active clock edge which can be either positive or negative. An asynchronous design is used to save device resources and to have faster response time for the system. This design method is relaying on propagation delays within a device. If the speed is increased in the system, the timing constraints can be broken and the design is not reusable [7].

In safety-critical systems it is preferable to use a synchronous design method in favour of an asynchronous. A more stable and predictable system is achieved and today’s FPGAs provide large amounts of gates, registers and memory, so there is no need to save resources. Usually there is no need getting as fast response-time as possible in safety-critical systems and the design will also be reusable.

4.3 Latches

Latches are elements that can hold a value until a new value is assigned. The latch is transparent which means that the input signal can propagate directly to the output. As a consequence of this, if a glitch appears at the input it can propagate directly to the output which is not desirable in safety-critical systems [7].
To avoid glitches, a flip-flop can be used instead because it is not transparent. Inferred latches are latches that are unintentional by the designer and are created by the synthesis tool to hold a value. To avoid these there are different coding techniques that can be followed. Inferred latches can appear e.g. when forgetting the “when others” in CASE-statements, or when forgetting the “else” in the IF-statement.

--- Example 4.1: Forgetting the “else” in IF-statement. ---

ENTITY example IS
Port (select, a, b, c: in std_logic;
     Out_d: out std_logic);
End entity example;

Architecture behaviour of example IS
    Variable x1, x2:std_logic;
Begin
    Process(a, b, c, select) IS
    Begin
        If (select = '1') then
            x1 := a and b;
            x2 := x1 xor c;
            Out_d <= x1 and x2;
        End if;
    End process;

The Select input will act as an enable for the latch as can be seen in figure 4.2. Another example of inferred latches is when forgetting to assign a default value to a signal at the beginning of a process.
4.4 Flip-flops

Using flip-flops is the preferred state-holding element to design with, but there are certain things to look out for. For the flip-flop to work correctly the input signal needs to be stable during a period of time before and after the clock edge.

When using flip-flops in the design it can be hazardous to trip on different edges in the same design; this can cause timing problems. The reason is that when looking at both the positive and negative edge there is a possibility to miss the edge.

To assume to have a perfect clock with 50% duty cycle is very optimistic, and this giving signals only half the clock cycle to propagate. In the real design, the actual time that is available can be much smaller [4].

4.5 Clocks

Clocks are causing most of the problems in designs. The designs usually contain multiple clocks, which can cause a mix-up between them. Therefore it is vital to document which clock the signal follows.

Example 4.2: Signals that are clocked by two different clocks.

```vhdl
SIGNAL a : STD_LOGIC; -- clk1, Comment
SIGNAL b : STD_LOGIC_VECTOR(31 DOWNTO 0); -- clk2, Comment
```

4.5.1 Clock domain crossing

According to [8] a clock domain is defined as that part of the design driven by either a single clock or clocks that has constant phase relationship. A clock and its inverted clock or its divided clocks are considered a clock domain. Clocks that have variable phase and time relationships are considered different clock domains.

When transferring data between clock domains, the risk of metastability is increased resulting in corruption of data. Therefore it is crucial to do a timing analysis when going between different clock domains.

4.5.2 Internally generated clocks

Internally generated clocks are clocks that are created with combinational logic in the code and this can cause timing problems in the design. The combinational logic can produce glitches and the delay, due to combinational logic, can cause timing problems. If this signal is used as an internal clock-signal, then the glitches and timing problems can cause metastability.

Internally generated clocks can also cause limited testability, because logic driven by the internally generated clock can not be part of a scan chain. Internally generated clocks also make it more difficult to constrain the design for synthesis.
4.6 State Machines

State machines are useful for representing logic that is controlling a sequence of events in a digital circuit. A FSM is designed to transition through a pattern of defined states.

4.5.3 Gated clocks in the RTL

According to [9] a gated clock is a clock signal that is enabled or disabled by another signal as can be seen in figure 4.3. By disabling the flip-flop, the power consumption is reduced and this is widely used in low power designs.

As can be seen in figure 4.4, gating clocks can cause glitches which can make the flip-flop clock at the wrong time and clock in the wrong data. Also, the skew of different local clocks can cause hold time violations. Gated clocks can cause limited testability because the logic clocked by a gated clock cannot be part of a scan chain. If gated clocks must be used in the design, it should be well documented.

4.6 State Machines

State machines are useful for representing logic that is controlling a sequence of events in a digital circuit. A FSM is designed to transition through a pattern of defined states.
There are different kinds of state machines and the most common are Mealy and Moore, where

- **Mealy machine’s outputs are a function of present state and all inputs.**
- **Moore machine’s outputs are only a function of the present state.**

If a SEU is hitting a memory cell, the state machine can be affected depending on the state coding.

According to [10] a mapped state is a state that is declared and this is also called a legal state. If the state is unmapped the state is not declared and is then called illegal.

There are different kinds of bit patterns and here are the most common.

### 4.6.1 Binary coding

In binary coding, the sequence for five states is: 000, 001, 010, 011 and 100. These states are legal while the others 101, 110, 111 are illegal. The states are shown in figure 4.5.

If e.g. a memory cell is hit by a SEU it is possible to reach an illegal state and then the state machine will freeze or have a strange behaviour. It is also possible to transition to a legal state on the influence of SEU, but to the wrong one. This can be very hazardous if the state is e.g. controlling a critical output.

**Example 4.3: Binary coding**

If we are in state 001 and get hit by a SEU, we can, for example transition to 011, which is a legal state, but the wrong one.
4.6.2 Gray coding

![Figure 4.6. Gray coding](image)

In gray coding there is a hamming distance of one, which means that only one bit is changed at a time between states.

--- Example 4.4: Gray coding ---
We can have the sequence: 000, 001, 011, 111, 110, 100 and the rest (101, 010) are illegal as can be seen in figure 4.6. If we are in state 011 and get hit by a SEU there are in this case three states that are possible to transition to. We can either transition to one of the two neighbouring legal states 001 and 111 or the illegal state 010.

It seems pretty straightforward to implement an algorithm for safe handling of the states. However, as can be seen in example 4.5 when there are more bits in the states, Gray coding needs a complementary safe detection logic to be safe against SEU.

--- Example 4.5: Gray coding ---
We can have the sequence: 0000 -> 0001 -> 0011 -> 0010 -> 0110 -> 0100 -> 1100 -> 1000 -> 0000. If we are in state 0000 and get hit by a SEU, then we can e.g. transition to state 0010, which is legal, but the wrong one. However this is not a neighbouring state as in the example before.

There have to be an even number of states for each closed cycle for gray-coded state machines to work properly. Gray coding is good if resources are to be saved or when designing low power systems, because it only changes one bit when transitioning between two states.

4.6.3 One-Hot coding

One-Hot is the fastest state encoding technique and can be good if a fast response time is needed. In One-Hot coding only one bit is high at a time. This state-
coding technique requires one flip-flop for every state, so therefore it is very area demanding. There are also a lot of illegal states. For five states the pattern can be 00001, 00010, 00100, 01000, 10000 and all other states are illegal as can be seen in figure 4.7.

According to [11] One-Hot coding is safe in the way that if e.g. a memory cell gets hit by a SEU, while in a legal state, the only transition is to an illegal state. Therefore the transition will never be to the wrong legal state. One reason not to use One-Hot for safety-critical applications would be that because it is so area demanding the probability of getting hit by a SEU is increased compared to other smaller area state-encodings. One-Hot together with TMR (Triple Mode Redundancy) voting can be good way to make the design safe.

TMR can be seen in figure 4.8. If one bit for one state is corrupted, the other two redundant bits will make sure that the state is still the right one. TMR is very easy to implement and can be a good solution for safety. However, if
implementing it wrong, it can get a glitchy behaviour for the TMR resulting in doing more damage than good. TMR is also very area demanding.

4.7 Synthesis tools

Today’s synthesis tools are geared towards timing and area optimization, which will reduce the necessary redundant logic for fault tolerance. The user really needs to know what the synthesis tool does; otherwise the implementation can change and be something the user did not want.

4.7.1 Safe attribute

The synthesis tools usually have a function called “safe” and are a demand from the aviation industry [12]. When using this function in a state machine, the tool will take care of all the illegal states at the same time. The user has to be careful when using this option, because it usually creates more gates than wanted.

Example 4.6: The “safe” attribute in VHDL.

```vhdl
TYPE states IS (s1, s2, s3, s4, s5)
SIGNAL current_state, next_state : states;
Attribute SAFE_FSM: Boolean;
Attribute SAFE_FSM of states: type is true;
```

If a binary encoded FSM flips into an illegal state, the safe option will return the FSM into a known state that is defined by the OTHERS statement.

If a binary encoded FSM flips into a good state, this error will go undetected. If the FSM is controlling a critical output, this could be very hazardous.
Chapter 5

Available rule checking tools for HDL

In this chapter, a market analysis of the available tools is presented.

5.1 General

A market analysis has been performed to gain knowledge of the tools that are available. The tools that have been studied can be seen in table 5.1. The tools have been selected on the basis on how common they are and how well they have been marketed. It is possible that some tools have been overlooked, but in general the most commonly used tools are represented here. These tools are very different in what is included in the tool and some are more advanced than others. However, they all have one thing in common; they are all static rule checking tools for HDL.

5.2 HDL Designer (Design checker)

Design Checker [13] is a static analysis and checking tool. It is a part of the HDL Designer series which is a complete design environment from Mentor Graphics. HDL Designer can be run in both Windows and UNIX and supports VHDL and Verilog.

5.2.1 User interface

Design checker is closely linked to the other programs in the HDL Designer series. It can be run with a fully GUI or in batch mode.

In batch mode commands can be written in the HDL Designer log window to control Design Checker. The scripting language used is Tcl (Tool command language) and it is possible to configure Design Checker from here, e.g. which rules to include in the run and which information the result file should contain.
Table 5.1. The tools that are included in the market analysis.

Design Checker is cross-referenced to HDL Designer DesignPad which is a code editor, and it is here the violations will be shown. It is also possible in HDL Designer to convert the code to graphics to easier understand how it works and help documenting the code. The code can be converted to Block diagram, State diagram and Flow chart.

5.2.2 Rules

The GUI supports a function to organize the rules into sets, and organize the sets into policies. With this function it is much easier to handle all the rules that are included.

The tool comes with predefined rules:

- **Base rules:** Basic rules for all rule-sets. These rules are read-only and can be copied to the user’s own rule-set and then be edited.
- **RMM ruleset:** Rules that follow the RTL coding guidelines which are described in [4].
- **Essential ruleset:** Rules that are typically used in the design community.
- **Altera and Xilinx ruleset:** Rules for making designs compatible with Altera and Xilinx tools.

The rule categories in Design Checker are:

- **Standard Language Syntax & Semantics:** Specifies the language or dialects that apply.
5.3 Discovery Verification Platform (LEDA)

- **Coding Practices**: Design style (e.g. synchronous or asynchronous, clocking scheme, gated clocks, language constructs and naming conventions).

- **Simulation**: Limited checking for design style issues that impact simulation (e.g. race conditions).

- **Synthesis**: Checks for constructs which are typically not synthesizable or other synthesis coding issues.

- **Testability**: Checks for coding styles (e.g. combinational loops), that impact Design-For-Test (DFT).

- **Portability, Reuse and Cross-language compatibility**: Checks which limit the constructs used in either language to a subset which can reliably be translated between them. These checks are important for re-usable components which need to support either language. These checks can also ensure that designs use a subset which is common to all tools used on the project.

There is a search function in Design checker to help the user to find the rules. Every rule has its own attributes with certain keywords that are searchable, so it is easy to find.

A way of measuring code quality is to have different levels of violations. The severity-levels can be custom made by the user in the severity-level tool, where the scoring and weighting can be changed together with the colour of the violations. The default severity-levels are Error, Warning and Note. These names can be changed and additional levels or less can be changed. The severity-level can be set for the whole rule-set, so score and weight does not need to be changed for every rule that is added to that rule-set.

5.2.3 Reports

The tool has also an extensive report function. The reports can tell the user how many rules have been violated and where in the code the violation is located. There is also a measurement of the code quality which depends on the severity of the violations.

When Design checker has finished analyzing the code, the reporting of violations is shown in DesignPad as a list of the violations with the line, severity and violation message. The violations are also shown in Design checker, in a result window were the violations are explained more in depth than in DesignPad. A summary window is also present were statistics of the different violations and rule categories are shown. It is possible to export both the result and the summary to CSV (Comma Separated Format), TSV (Tab Separated Format) and HTML (Hyper Text Mark-up Language) for further evaluation.

5.3 Discovery Verification Platform (LEDA)

LEDA [14] is a programmable mixed language HDL design/coding style checker and is part of the Discovery Verification Platform from Synopsys.
LEDA consists of two tools that are sold separately. The first tool is a specifier for creating and compiling custom rules and to manage rulesets and coding standards. The second tool is a code checking tool that can check VHDL and Verilog code.

5.3.1 User interface

LEDA can be run in a GUI, in Batch mode or in Tcl Shell Mode. The batch mode and GUI modes are linked together so the user can switch between them when doing different tasks. When in GUI mode the Tcl Shell mode is available at the bottom of the main window. The Tcl Shell can also be used for manage the LEDA projects, rule configurations, and runs with the checker.

5.3.2 Rules

There are four different types of rule categories:

- **Block level rules** (Coding rules): Unit-wide (e.g. detect multiple clocks in an architecture.)
- **Chip level rules** (Hardware rules): Design-wide (e.g. all flip-flops in the design must be active on the rising edge.)
- **Netlist rules** (Design rules): Rules that run on the gate-level netlist for the design (e.g. control signal crossing clock domain without data transfer.)
- **SDC rules**: Rules that check SDC (Synopsys Design Constraint) files for internal consistency and consistency with the design that they constrain.

It is possible to create completely new rules using the LEDA Specifier. The user must use templates which are models of how the code should appear (e.g. what HDL constructs should or should not be present, in what order, and so forth.) The user can write design rules using the templates listed below. Other templates, such as clock and synchronous reset, also contain attributes that can be used to write chip-level rules:

- Design Template
- Connectivity Template
- Test Signal Template
- Data Signal Template
- Flip-flop Template
- Latch Template
5.3.3  Reports

After the run of the checker, there is information about the complete hierarchy of the design, and a summary panel that reports the number and kind of violations that are found. An Error viewer are also present were the user can learn more about the violations using a HTML-based help system. There is also a Path Viewer and Clock and Reset Tree browser that help the user to visualize errors for rules where tracing information is available. It is also possible to jump from the Error Viewer to the exact location in the code where the violation exists.

5.4  Verification Navigator (VN-Check)

VN-Check [15] is a built-in rule checking tool in Verification Navigator from TransEDA. The company is a provider of coverage and verification measurement solutions for electronic designs. The tool supports VHDL, Verilog and SystemVerilog.

5.4.1  User interface

VN-Check may be run using a GUI or in batch mode using a CLI (Command Line Interface).

5.4.2  Rules

The rules can be organized into five categories:

- **Coding rules**: Specifies how coding constructs must be used. E.g. are Set/reset related problems, Combinational Feedback loops, Clock Domain Crossings, Size matching problems, Signals Read/Written problems, Latch inference and Register related problems.

- **Style rules**: Refer to things such as allowable number of modules per file, statements per line, etc.

- **Documentation rules**: Checks that constructs in the code are consistently commented to ensure code readability and maintainability.

- **Naming rules**: Can be used for developing a naming convention across a design hierarchy.

- **Verify rules**: For use with SystemVerilog.

From these categories, the rules database is then built.
The Rules Databases supplied with VN-Check are:

- **Best Practices**: Contains rules that represent generally accepted good coding practice.

- **RMM**: Contains a set of rules related to [4].

- **Synthlite**: Contains rules that check for compliance with the basic requirements of logic synthesis tools.

- **Synth**: Contains a comprehensive set of rules related to logic synthesis.

- **verilog2vhdl**: Checks for compliance with a set of rules that will permit easy translation of a Verilog model to VHDL.

- **vhdl2verilog**: Checks for compliance with a set of rules that will permit easy translation of a VHDL model to Verilog.

- **DFT**: Based on standard best practices for implementing testable designs. Documents such as the test standard developed by the VSIA (Virtual Socket Interface Alliance) provide the basis for this rule set.

- **FSM**: Provides a set of rules related to the design of finite state machines.

- **FPGA**: Provides a set of rules based on the FPGA OpenMORE extension of the RMM. Note that the rules are not targeted at any individual device family or device vendor.

- **verilog2systemverilog**: Checks for compliance with a set of rules that will permit easy translation of a Verilog model to a model that uses SystemVerilog specific constructs.

- **OpenMORE**: Implements the OpenMore scoring system.

- **SVAH**: Includes SystemVerilog Assertion related rules, and also assertion-density rules that provide assistance about where to add assertions in the design.

The rules databases can be reconfigured or the user can define its own rules database and choose from the different rules that are supplied with the tool. There are over 500 rules for VHDL and over 600 for Verilog. The creation of an entirely new rules database can be done in the rule-database editor that is supplied with the tool. There is a search function in VN-Check to look for rules; this is very convenient when creating a custom rules database.
5.4.3 Reports

The output from VN-Check consists of reports that list any rule violations and identify the lines of HDL source that caused the violations. If the OpenMORE scoring system is also selected for use, the score for the design will also be reported.

To generate a result file, the user has to be in the CLI mode and write the appropriate command. The reports can be generated as ASCII text or as HTML for examination using a browser. When the user is in the GUI mode he/she can view the results in the VN-Check Results Viewer. FSM’s in the code can be graphically displayed in the result so it will be easier to see what the problem is. The FSM can be changed in a special editor, and then the code will be updated.

5.5 Riviera-PRO

Riviera-PRO [16] is a VHDL and Verilog simulation environment from Aldec Inc. It includes a VHDL/Verilog compiler, PSL (Property Specification Language)/OVA (OpenVera)/SVA (SystemVerilog Assertions) compiler, HDL/SystemC/Assertion simulation engine, coverage engine, advanced debugging tools and Lint.

5.5.1 User interface

Riviera-PRO can operate in three modes: GUI-mode, Command Line Mode and Batch mode. For VHDL, Riviera-PRO has Basic Lint and for Verilog it has both basic and advanced Lint. The Lint tool is integrated in the VHDL compiler. Options for the VHDL Lint can be set in the GUI.

5.5.2 Rules

For VHDL basic Lint is offered by the tool and this Lint is built-in and integrated into the VHDL compiler. For Verilog both basic Lint and also advanced Lint are available and are a selection of rules from the STARC RTL Design Style Guide for Verilog HDL.

For VHDL the rule categories are:

- Coding Styles
- DFT
- Design Style
- Language Construct
- Simulation
- Synthesis
The only options available for Lint checking in VHDL are to select which rule categories the tool should check the code against, so there are no options for any modification of the rules.

5.5.3 Reports
The result is reported when compiling the code in HTML, CSV or ASCII text.

5.6 More tools
Due to the difficulty of attaining documentation, the following tools are not that thoroughly presented as the former.

5.6.1 Incisive Enterprise Simulation (HDL Analysis)
Incisive Enterprise Simulation [17] is a verification environment that includes simulation, static and dynamic assertion checking, HDL analysis and code coverage analysis. The tool HDL analysis has over 500 rules for VHDL and Verilog included. The rules are for synthesizability, race conditions, code reusability, clock domain synchronization, FSM coding and are based on RMM. It also has a graphical interface to sort, filter and analyze messages with source code.

5.6.2 Spyglass
Spyglass [18] is a static and dynamic analysis tool for RTL from Atrenta. Spyglass consists of different programs (DFT, CDC, Constraints and Power). CDC handles Clock Domain Crossings, DFT handles Design For Test issues, Constraints handles constraints and Power handles the power consumption. The Spyglass program has access to RTL analysis against a set of rules. It can check coding, style, synthesis, simulation, verification, FSM, clock and reset issues.

The tool is available for UNIX and supports Verilog, VHDL, SystemVerilog and mixed language designs.

5.6.3 VLint
VLint [19] from Veritools is an RTL code checker that is available for UNIX and Windows and supports Verilog. The tool can check for issues in the code such as (Race detection, Clock domain boundary check, RMM, Synthesis checks, Testability checks and Net-list checks). It is possible to communicate with VLint using Tcl or Perl scripts. The tool can operate in a GUI and in batch mode.

5.7 Summary
The different capabilities of the tools can be seen in table 5.2. It was difficult to obtain documentation for the three tools to the right in the table. Therefore
### 5.7 Summary

<table>
<thead>
<tr>
<th></th>
<th>Design Checker</th>
<th>LEDA VN-Check</th>
<th>Riviera-PRO</th>
<th>HDL-Analysis</th>
<th>Spyglass</th>
<th>Vlint</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Unix</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VHDL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Verilog</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>User interface</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GUI</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>API</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Tcl support</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>Perl support</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Rules</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>Naming</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>Coding</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Portability</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Reusability</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>Synthesizability</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Reports</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSV</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TSV</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HTML</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ASCII text</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

| **Table 5.2.** Summary of the different tools and their capabilities. |

some of its capabilities may have been overlooked and is marked with -. Saab is interested in a tool that supports VHDL. Therefore VLint is ruled out at this point because it is the only tool that does not support VHDL. Furthermore, Riviera-PRO seems too simple for the current needs, because of the lack of parameterization and the type of rules that are included looks very simple for VHDL. However, for Verilog this tool seem more advanced. The rest of the tools seem very capable and could work well for Saab.

Design Checker from Mentor Graphics was chosen, since Saab already uses their products. Then VN-Check from TransEda was chosen to compare with. It was no problem to get an evaluation license to VN-Check for evaluation. Both Design Checker and VN-Check supports Windows which is the preferred choice for Saab.

Spyglass from Atrenta, HDL analysis from Cadence and Leda from Synopsys would be strong competitors to Design Checker and VN-Check, but they only supports UNIX and it did not seem like they offered any evaluation copies. Also there is a need to restrict the evaluation to two tools, because of the time limit and the scope of this thesis.
Chapter 6

Evaluation of rule checking tools

This chapter describes the different evaluation areas and also introduces a test code.

6.1 General

The evaluation of the tools will look more into the following aspects.

- **General applicability:** How complicated is the tool to use? Is it possible to implement the more simple rules and how are they implemented? Are there different severity levels? What can be said about the result, are the explanations of the violations intuitive and is there any quality score?

- **Specific applicability:** Is it possible to implement the more advanced, safety-critical rules in the tool and how is it done?

- **Reliability:** A code with a wide variety of already predefined faults will be tested by the tool. The two tools will later be compared if they find the faults or not. This is not a complete performance test of the tools, but will work as an indication of what the tools are able to do.

6.2 Testplan

The rules are an extract from Saab’s own rules and they cover most of the functionality wanted in a tool. The test plan can be found in table 6.1

The √ in the column for every tool means that the rule has been implemented in the tool. The * means that the rule is partly implemented. Finally - means that the rule has not been available in the tool and therefore not implemented.
### Table 6.1. Testplan for evaluation of tools.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Category</th>
<th>Design Checker</th>
<th>VN-Checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Prefixes and Suffixes</td>
<td>Documentation</td>
<td>✓</td>
<td>∗</td>
</tr>
<tr>
<td>2. Lowercase and Uppercase</td>
<td>Documentation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3. Number of characters in names no more than 25</td>
<td>Documentation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4. Only one VHDL statement per line</td>
<td>Documentation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5. Line length max 93 characters</td>
<td>Documentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Indentation with 2 spaces</td>
<td>Documentation</td>
<td></td>
<td>∗</td>
</tr>
<tr>
<td>7. Port-mapping has new lines for each parameter</td>
<td>Documentation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>8. Using rising_edge and falling_edge for the clock</td>
<td>Clock</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>9. Either Mealy or Moore state-machines are used in the design</td>
<td>FSM</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>10. FSM shall be created with one sequential process or a combination of two processes, one sequential process that controls the state transitions and one combinational process to control the dataflow</td>
<td>FSM</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>11. Buses shall be declared with bits given from MSB to LSB</td>
<td>Coding</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>12. Latches in the design.</td>
<td>State-Holding elements</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>14. Gated clocks in the RTL.</td>
<td>Clock</td>
<td>✓</td>
<td>∗</td>
</tr>
<tr>
<td>15. Internally generated clocks in the design.</td>
<td>Clock</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>16. Clock domain crossings.</td>
<td>Clock</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>17. Internally generated resets.</td>
<td>Reset</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>18. Asynchronous FSMs are not allowed.</td>
<td>FSM</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>19. All outputs from an FSM have to be initiated asynchronously at reset.</td>
<td>FSM</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>20. All states can transition to a known state</td>
<td>FSM</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>21. Gray coding</td>
<td>FSM</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>22. One hot coding</td>
<td>FSM</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### 6.2.1 Motivation for the more simple rules

1-7. These rules make the code look better and it can contribute to a more understandable code. If the code is messy and many of these rules are violated, errors can easily be foreseen that can cause e.g. logic errors.

8. There are several ways of describing the clock. It can be good to stick to one of the descriptions, so the code will be more uniform.

**Example 1**

If `rising_edge(clk)` then

**Example 2**

If `clk’event and clk='1'` then

In this case `rising_edge` or `falling_edge` is preferred, because it is easier to see what type of edge is used.
9. Stick to one representation of FSMs. Moore is the preferred choice because Mealy can cause timing problems. Mixing them may also cause other problems.

10. This rule makes the code look more structured and it is easier to see how the FSM works.

6.2.2 Motivation for the more advanced rules

11. This is a very important rule. E.g. if the bus in one module is declared with MSB (Most Significant Bit) to LSB (Least Significant Bit) and another bus with MSB to LSB, there is an inversion. This will probably show later in the simulation, but if found early, time can be saved. It is also good for the codes reusability, to have this rule.

12. The preferred state-holding elements are flip-flops, because of transparency with latches. To save resources, reduce the number of unnecessary latches.

13. Do not clock on different edges because of possible timing problems.

14. Avoid gated clocks if possible, when these can cause timing problems.

15. Internally generated clocks should be avoided because of possible timing problems.

16. This problem can cause timing problems if it occurs.

17. Internally generated resets should be avoided; all registers in the design should be reset at the same time. Otherwise analysis and design will be much more difficult.

18. Asynchronous FSMs should be avoided at all costs, because of reusability-and timing problems.

19. Always have an asynchronous reset of all the registers. It is also important to initialize the outputs at reset to avoid putting unknown values to the outputs.

20. If it is not possible to go to a known state from all states, the design can freeze.

21. If Gray coding is used, the user needs to be enlightened that it is not sufficient to make Gray coding in a tool, it should be made directly in the HDL code. Also there have to be an even amount of states for each cycle for gray coding to work correctly.

22. If one-hot coding is used, the user needs to be enlightened that an error detection mechanism should be used (like TMR).
6.3 Testcode

In order to test the tools for the different rules, a code with already implemented faults is needed. The code that is used is called TEST_PROJECT and is a code that will only work as an example and do not have any real functionality. The project consists of five different codes that will cover most of the rules. First is the TEST_CODE_entity.vhd which is a port mapping code and is the top-level. Then there are two FSM codes, one with Moore and one Mealy FSM. There is also a CDC code and a gated clock code. Finally there is an internal clock code. The internal clock code has a connection with the gated clock code. A clock is generated in the gated clock code and is then used by the internal clock code. The project is run by two different clocks, to show CDC. The TEST_PROJECT can be seen in a block diagram in figure 6.1.

The code for the TEST_PROJECT can be found in Appendix C. It should be mentioned that this code is very simple and will only show some basic faults in a code.
The following intentional faults are implemented in the TEST_PROJECT code:

**TEST_CODE_entity.vhd**
- The filename is not written consistent in Uppercase.
- Signal D1 should be written with lowercase to be consistent with other signals.
- Instances should be written like U#_* instead of U*, where the star is the name of the instance.

**GATED_COMP_entity.vhd**
- The filename is not written consistent in Uppercase.
- The entity is not written consistent in Uppercase.
- To be consistent there should be only one port mapping declaration per line.
- The architecture is not written consistent in Uppercase.
- There is a gated clock in the code.

**INTERNAL_COMP_entity.vhd**
- The filename is not written consistent in Uppercase.
- To be consistent there should be only one port mapping declaration per line.
- There is an internally generated clock in the code.

**CDC_COMP_entity.vhd**
- The filename is not written consistent in Uppercase.
- There is a CDC in the code.

**FSM_MOORE_entity.vhd**
- The filename is not written consistent in Uppercase.
- Signal D1 should be written with lowercase to be consistent with other signals.
- The d_out bus is not declared in a consistent way.
- There is a mixing of rising_edge and falling_edge in the design.
- There is state that do not transition to a known state.
- Signal D1 has no initial value and therefore an inferred latch.
- No when others in case statement and therefore inferred latch.
**FSM_MEALY_entity.vhd**

- The filename is not written consistent in Uppercase.
- Signal D1 should be written with lowercase to be consistent with other signals.
- Mealy FSM is used. Moore FSM is preferred.
- There is a CDC in the code.
- No when others in case statement and therefore inferred latch.

### 6.4 Avionics code

The other codes will be real code from an aviation project at Saab Avitronics. There will be one code that is already tested manually and one code that has not yet been tested. These codes are restricted, so the codes will not be a part of this thesis. Information about which rules are violated are not shown, because it can reveal things that would not benefit the company.

Only different categories of the violations will be shown, to see where most of the violations exist. The aviation code is only checked with Design Checker due to setup problems with VN-Check, but it gives a hint on what kind of violations there are in the code (This code will not act as an evaluation criterion, when comparing the two tools).
Chapter 7

HDL Designer (Design checker)

In this chapter Design Checker is evaluated.

7.1 General applicability

7.1.1 Running the tool

To start using Design Checker, open the projects and files in Design Manager, then run Design Checker and there choose which rules and policies to use in the check. After this the check can be started. When the check is finished, a result is presented that is also cross-referenced to Design Pad where the code can be edited.

7.1.2 Rules

The rules are parameterized and can in some degree be changed to suit the users own rules. Figure 7.1 shows an example of rule properties.

There are certain parameters that are common to all base rules.

- **Name**: A describing name for the rule that is unique.
- **Severity**: Sets the severity level for the rule.
- **Language**: Specifies what language the rule is applied for.
- **Hint**: Some additional information of what could have caused the violation.
- **Short Description**: Describes the rule in more detail.
- **Keywords**: A searchable keyword so the rule will be easy to find.
There are additional parameters to every rule category that can include e.g. “pattern” where a regular expression can be written to match a special pattern in the code. There is also “applies to” which define what constructs the rule should apply to.

If the rules could be implemented and how it is done can be seen in table 7.1.

### 7.1.3 Severity levels

The severity levels for Saabs rules are High, Medium and Low. There are also guidelines.

The total score will be the weighting multiplied with the score. Colours can be chosen, so it is easy to see the difference between the severity levels.

### 7.1.4 Result

The categories for the summary window are:

- **Settings**: Displays the active policy, library, primary and secondary design units, the master clock and reset signals, the depth of the analysis and the check level.

- **Exclusions**: Shows the exclusion settings. It states the number of Policy Disabled rules, code/rule exclusions, black box files, and exclusion pragmas.

- **Design quality**: The quality is measured in percentage and the different rules categories that are used have a quality percentage. 100% means that all the rules in that category are fulfilled.

---

<table>
<thead>
<tr>
<th>Parameters of Configured Rule: 4.03.01 - Avoid Gated clocks in RTL</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>4.03.01 - Avoid Gated clocks in RTL</td>
</tr>
<tr>
<td>Severity</td>
<td>Guideline</td>
</tr>
<tr>
<td>Score</td>
<td>2</td>
</tr>
<tr>
<td>Weight</td>
<td>1</td>
</tr>
<tr>
<td>Language</td>
<td>VHDL Any, Verilog Any</td>
</tr>
<tr>
<td>Hint</td>
<td>Avoid gated clocks in the RTL</td>
</tr>
<tr>
<td>Short Description</td>
<td>Checks for gated clocks</td>
</tr>
<tr>
<td>Keywords</td>
<td>clocks, gated, gating, synchronous, enable, load, register</td>
</tr>
<tr>
<td>Action</td>
<td>Disallow All Gated Clocks</td>
</tr>
</tbody>
</table>

Figure 7.1. Parameters for one of the RMM rules.
### 7.1 General applicability

<table>
<thead>
<tr>
<th>Rule</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Prefixes and Suffixes</td>
<td>Use a rule called “naming” which is a base rule and can be found in the naming directory. This rule has support for regular- and simple expressions. To match the prefix for variable v_ write v_* which will match any string that begin with v_.</td>
</tr>
<tr>
<td>2. Lowercase and uppercase</td>
<td>Use the rule “naming” here. Lowercase and uppercase is predefined for this rule, so choose either one of them in the “Pattern” field.</td>
</tr>
<tr>
<td>3. Number of characters in names no more than 25</td>
<td>Use the rule “Length” in the style directory and modify the max length to 25 and to what types it should apply to.</td>
</tr>
<tr>
<td>4. Only one VHDL-statement per line</td>
<td>Use the rule “Use a separate line for each statement” which can be found in Basic rules in the RMM ruleset. This rule will check that it is only one statement per line. Declarations are excluded unless there is a declaration on the same line as another statement.</td>
</tr>
<tr>
<td>5. Line length with max 93 characters</td>
<td>Use the rule “Line length less than 72” in the Basic rules directory in the RMM ruleset. Change the maximum characters from 72 to 93.</td>
</tr>
<tr>
<td>6. Indentation with 2 spaces</td>
<td>There are no rules and it is not possible to parameterize any of the rules to check this item. However it can be achieved using DesignPad where the indentation space length can be set and also run an auto indentation on the code.</td>
</tr>
<tr>
<td>7. Port-mapping has new lines for each parameter</td>
<td>Use the rule “Declaration style” in the style directory. This rule will make sure that it is only one port declaration on each line.</td>
</tr>
<tr>
<td>8. Using rising_edge+ and falling_edge for the clock</td>
<td>Use the base rule “Flip Flop Edge Triggers” in the registers directory. This rule can check that only positive or negative edge triggered flip-flops are used.</td>
</tr>
<tr>
<td>9. Either Mealy or Moore state-machines are used in the design</td>
<td>Use the base rule “FSM Coding Style” in the FSM directory. Choose the output style to Mealy or Moore.</td>
</tr>
<tr>
<td>10. FSM shall be created with one sequential process or a combination of two processes, one sequential process that controls the state transitions and one combinational process to control the dataflow</td>
<td>In the base rule “FSM Coding Style”, choose coding style, Clocked + Separate Transition/output, or clocked + combined Transition/output.</td>
</tr>
</tbody>
</table>

Table 7.1. General applicability with Design Checker.
Figure 7.2. Typical design quality report in Design Checker.

- **Violations:** Shows how many violations there are from the different severity levels, and in which segments the violations exist. The different segments are file, entity, architecture, module, package body, package header, configuration and unknown.

- **Rules:** Shows the name of the current default policy used for analysis. Also shows the number of failed checks.

- **Design units:** Displays a list of failed checks.

In the result it is easy to see what rules caused the violations and in what areas. The total quality score is also a nice feature. In figure 7.2, a typical design quality report from Design Checker is shown. In the column ruleset, the different rulesets in a tree structure is shown. In the score column, the total score for that
ruleset is presented. The score is based on the different severity levels for the rules. Also a percentage column that is calculated from the score column is presented. In this case, the severity levels are rule and guideline and they have each separate columns that show how many of the rules have failed.

7.2 Specific applicability

7.2.1 Rules

If the rules can be implemented and how it is done is shown in table 7.2.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>11. Buses shall be declared with bits given from MSB to LSB</td>
<td>In the RMM rulesets, and in general naming conventions, there is a rule “Descending bus order” which will do here.</td>
</tr>
<tr>
<td>12. Latches in the design.</td>
<td>This can be checked using the base rule “Latch Inference” which can be found in the registers directory. This rule will check for latches in the design.</td>
</tr>
<tr>
<td>13. Mixed positive- and negative-edge triggered flip-flops.</td>
<td>This can be found in the Clocks &amp; Resets folder in the RMM ruleset.</td>
</tr>
<tr>
<td>14. Gated clocks in the RTL.</td>
<td>This can be checked using the base rule “Gated Clocks” in the Clocks &amp; Resets directory.</td>
</tr>
<tr>
<td>15. Internally generated clocks in the design.</td>
<td>This can be checked using the base rule “Internally Generated Clocks” in the Clocks &amp; Resets directory.</td>
</tr>
<tr>
<td>16. Clock domain crossings.</td>
<td>There is no capability in Design checker to look for CDC errors. However this can be done with Mentor Graphics tool 0-In CDC.</td>
</tr>
<tr>
<td>17. Internally generated resets.</td>
<td>This can be checked using the base rule “Internally Generated Resets” in the Clocks &amp; Resets directory.</td>
</tr>
<tr>
<td>18. Asynchronous FSMs</td>
<td>Not available.</td>
</tr>
<tr>
<td>19. All outputs from an FSM have to be initiated asynchronously at reset</td>
<td>Not available.</td>
</tr>
<tr>
<td>20. All states can transition to a known state</td>
<td>Use the base rule “FSM Transition” in the FSM directory. This rule checks that unused states can transition to a recovery state and that all states transitions to a known state.</td>
</tr>
<tr>
<td>21. Gray coding</td>
<td>There are no rules available for this, but it may be implemented with script.</td>
</tr>
<tr>
<td>22. One hot coding</td>
<td>There are no rules available for this, but it may be implemented with script.</td>
</tr>
</tbody>
</table>

Table 7.2. Specific applicability with Design Checker.

7.2.2 Implement specific rules

If the rules that wanted is not available in the already predefined rules, there is a possibility to write scripts in HDL Design Pad. The scripts need to be written in Tcl and then run in HDL Designer log window. The scripts can be a pattern or string search in the code.
Example 7.1: Graycode in Design Checker

It would be too difficult to see if the user is using Gray coded state-machines and the user needs to write a comment that Gray coding is used. There is an option in the synthesis tool from Synplify, that if Gray code is used, a box can be checked in the synthesis tool that we want to use safe Gray-code. The tool will then in the code write that we are using safe Gray. It can look like this:

```
-- Optional type_encoding style scheme for Synplify
ATTRIBUTE syn_encoding OF current_state: SIGNAL IS "safe, gray";
```

Then we only need to search for the word Gray in the code, comments can be excluded from the search, to reduce the risk for flagging innocent comments. To search for Gray in the code, the following Tcl script is used:

```
# A program that finds the word gray (not case sensitive) in the code.

#Imports the vhd file
set fileId [open ./HDS/my_project/my_project_lib/hdl/gated_clocks_entity.vhd]
set z 0 # Counter for number of rows.
while {[gets $fileId line] >=0} {
    incr z
    set x [regexp -nocase gray $line] # Looks for the word gray (not case sensitive)
    if {$x > 0} {puts stdout "The word gray is found on row $z"} # Output
}
```

The script is not case sensitive, so all forms of gray will be found (Gray, GRAY, gray-coding etc.)
7.3 Reliability

After the tool is used on the TEST_PROJECT with the rules according to Appendix A, the following results are presented.

<table>
<thead>
<tr>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Guideline</td>
<td>File &quot;TEST_CODE_entity.vhd&quot; violates naming convention: use &quot;&lt;Lowercase&gt;&quot; for entity, architecture file names</td>
</tr>
<tr>
<td>30</td>
<td>Low</td>
<td>Signal &quot;d1&quot; violates naming convention: use &quot;&lt;Lowercase&gt;&quot; for signal names</td>
</tr>
<tr>
<td>33</td>
<td>High</td>
<td>Instance name &quot;U1&quot; violates naming convention: use &quot;U#_*&quot; for labelling instances</td>
</tr>
<tr>
<td>33</td>
<td>High</td>
<td>Internal generation of clock signal &quot;clk_c&quot; output from instance &quot;U1&quot;</td>
</tr>
<tr>
<td>34</td>
<td>High</td>
<td>Internal generation of clock signal &quot;clk_c&quot; output from instance &quot;U1&quot;</td>
</tr>
<tr>
<td>34</td>
<td>High</td>
<td>Instance name &quot;U2&quot; violates naming convention: use &quot;U#_*&quot; for labelling instances</td>
</tr>
<tr>
<td>35</td>
<td>High</td>
<td>Internally generated clock &quot;clk_c&quot; drives instance &quot;U2&quot;</td>
</tr>
<tr>
<td>36</td>
<td>High</td>
<td>Internally generated clock &quot;clk_c&quot; drives instance &quot;U2&quot;</td>
</tr>
<tr>
<td>37</td>
<td>High</td>
<td>Instance name &quot;U3&quot; violates naming convention: use &quot;U#_*&quot; for labelling instances</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3. TEST_CODE_entity.vhd violation report.

Violations from the file TEST_CODE_entity.vhd can be seen in table 7.3. Most of the violations are related to naming conventions. There are also some internally generated clocks.

<table>
<thead>
<tr>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Guideline</td>
<td>File &quot;INTERNAL_COMP_entity.vhd&quot; violates naming convention: use &quot;&lt;Lowercase&gt;&quot; for entity, architecture file names</td>
</tr>
<tr>
<td>11</td>
<td>Low</td>
<td>Multiple ports declared on one line, declarations should be on separate lines</td>
</tr>
<tr>
<td>19</td>
<td>Guideline</td>
<td>Use an externally controlled asynchronous reset signal to initialize all registers in &quot;test_code.U2&quot;</td>
</tr>
<tr>
<td>20</td>
<td>Guideline</td>
<td>Register &quot;out2&quot; has not been initialized using an externally controlled asynchronous reset signal</td>
</tr>
</tbody>
</table>

Table 7.4. INTERNAL_COMP_entity.vhd violation report.

A gated clock has been found, which can be seen in table 7.5.
<table>
<thead>
<tr>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Guideline</td>
<td>File &quot;GATED_COMP_entity.vhd&quot; violates naming convention: use &quot;&lt;Lowercase&gt;&quot; for entity, architecture file names</td>
</tr>
<tr>
<td>10</td>
<td>Low</td>
<td>Design unit &quot;gated_comp&quot; violates naming convention: use &quot;&lt;Uppercase&gt;&quot; for primary unit names</td>
</tr>
<tr>
<td>11</td>
<td>Low</td>
<td>Line length exceeds maximum of &quot;93&quot; characters per HDL line</td>
</tr>
<tr>
<td>15</td>
<td>Low</td>
<td>Architecture &quot;rtl&quot; violates naming convention: use &quot;&lt;Uppercase&gt;&quot; for architecture names</td>
</tr>
<tr>
<td>17</td>
<td>Guideline</td>
<td>Sensitivity list of process/always statement &quot;&lt;anonymous&gt;&quot; includes unneeded signal &quot;enable&quot;</td>
</tr>
<tr>
<td>19</td>
<td>High</td>
<td>Gated clock signal &quot;clk&quot; used to drive synchronous block &quot;&lt;anonymous&gt;&quot;</td>
</tr>
<tr>
<td>19</td>
<td>High</td>
<td>Derivation of clock gating through signal &quot;clk&quot;</td>
</tr>
<tr>
<td>19</td>
<td>Guideline</td>
<td>Use an externally controlled asynchronous reset signal to initialize all registers in &quot;test_code.U1&quot;</td>
</tr>
<tr>
<td>20</td>
<td>Guideline</td>
<td>Register &quot;clk_c&quot; has not been initialized using an externally controlled asynchronous reset signal</td>
</tr>
</tbody>
</table>

Table 7.5. GATED_COMP_entity.vhd violation report.

<table>
<thead>
<tr>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Guideline</td>
<td>File &quot;CDC_COMP_entity.vhd&quot; violates naming convention: use &quot;&lt;Lowercase&gt;&quot; for entity, architecture file names</td>
</tr>
<tr>
<td>22</td>
<td>Guideline</td>
<td>Use an externally controlled asynchronous reset signal to initialize all registers in &quot;test_code.U3&quot;</td>
</tr>
<tr>
<td>23</td>
<td>Guideline</td>
<td>Register &quot;sig1&quot; has not been initialized using an externally controlled asynchronous reset signal</td>
</tr>
<tr>
<td>29</td>
<td>Guideline</td>
<td>Use an externally controlled asynchronous reset signal to initialize all registers in &quot;test_code.U3&quot;</td>
</tr>
<tr>
<td>30</td>
<td>Guideline</td>
<td>Register &quot;out1&quot; has not been initialized using an externally controlled asynchronous reset signal</td>
</tr>
</tbody>
</table>

Table 7.6. CDC_COMP_entity.vhd violation report.
## Table 7.7. FSM_MOORE_COMP_entity.vhd violation report.

<table>
<thead>
<tr>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Guideline</td>
<td>File &quot;FSM_MOORE_COMP_entity.vhd&quot; violates naming convention: use &quot;&lt;Lowercase&gt;&quot; for entity, architecture file names</td>
</tr>
<tr>
<td>14</td>
<td>Low</td>
<td>Port &quot;D1&quot; violates naming convention: use &quot;&lt;Lowercase&gt;&quot; for port names</td>
</tr>
<tr>
<td>14</td>
<td>Low</td>
<td>Port &quot;D1&quot; violates naming convention: use &quot;&lt;Lowercase&gt;&quot; for names of ports: mode out</td>
</tr>
<tr>
<td>15</td>
<td>Medium</td>
<td>Dimension/Range definition &quot;(0 to 3)&quot;, for &quot;d_out&quot;, does not comply to descending order convention</td>
</tr>
<tr>
<td>20</td>
<td>High</td>
<td>FSM state variable &quot;state&quot; violates naming convention: use &quot;*_current_state&quot; for the current state</td>
</tr>
<tr>
<td>20</td>
<td>High</td>
<td>FSM state variable &quot;state&quot; violates naming convention: use &quot;*_next_state&quot; for the next state</td>
</tr>
<tr>
<td>25</td>
<td>Guideline</td>
<td>Incomplete sensitivity list for process/always statement &quot;&lt;anonymous&gt;&quot;. Signal &quot;reset&quot; is needed</td>
</tr>
<tr>
<td>29</td>
<td>Guideline</td>
<td>Use an externally controlled synchronous reset signal to initialize all registers in &quot;test_code.U4&quot;</td>
</tr>
<tr>
<td>49</td>
<td>Guideline</td>
<td>Register &quot;D1&quot; has not been initialized using an externally controlled asynchronous reset signal</td>
</tr>
<tr>
<td>61</td>
<td>High</td>
<td>Latch Inferred: Latch is created for signal &quot;D1&quot; for not being assigned values at all input conditions</td>
</tr>
<tr>
<td>62</td>
<td>Medium</td>
<td>Avoid using hard coded numeric values such as &quot;0101&quot; for literals used in assignment expressions</td>
</tr>
<tr>
<td>62</td>
<td>Low</td>
<td>Line length exceeds maximum of &quot;93&quot; characters per HDL line</td>
</tr>
<tr>
<td>64</td>
<td>Medium</td>
<td>Avoid using hard coded numeric values such as &quot;0110&quot; for literals used in assignment expressions</td>
</tr>
<tr>
<td>65</td>
<td>Medium</td>
<td>Avoid using hard coded numeric values such as &quot;1010&quot; for literals used in assignment expressions</td>
</tr>
<tr>
<td>66</td>
<td>Medium</td>
<td>Avoid using hard coded numeric values such as &quot;1001&quot; for literals used in assignment expressions</td>
</tr>
<tr>
<td>Line</td>
<td>Severity</td>
<td>Message</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>1</td>
<td>Guideline</td>
<td>File &quot;FSM_MEALY_COMP_entity.vhd&quot; violates naming convention: use &quot;&lt;Lowercase&gt;*&quot; for entity, architecture file names</td>
</tr>
<tr>
<td>14</td>
<td>Low</td>
<td>Port &quot;D1&quot; violates naming convention: use &quot;&lt;Lowercase&gt;*&quot; for port names</td>
</tr>
<tr>
<td>14</td>
<td>Low</td>
<td>Port &quot;D1&quot; violates naming convention: use &quot;&lt;Lowercase&gt;*&quot; for names of ports: mode out</td>
</tr>
<tr>
<td>22</td>
<td>High</td>
<td>FSM state variable &quot;state&quot; violates naming convention: use &quot;*_current_state&quot; for the current state</td>
</tr>
<tr>
<td>22</td>
<td>High</td>
<td>FSM state variable &quot;state&quot; violates naming convention: use &quot;*_next_state&quot; for the next state</td>
</tr>
<tr>
<td>31</td>
<td>Guideline</td>
<td>Use an externally controlled asynchronous reset signal to initialize all registers in &quot;test_code.U5&quot;</td>
</tr>
<tr>
<td>32</td>
<td>High</td>
<td>Use Moore style instead of Mealy style state machines</td>
</tr>
<tr>
<td>35</td>
<td>Medium</td>
<td>Avoid using hard coded numeric values such as &quot;1001*&quot; for literals used in assignment expressions</td>
</tr>
<tr>
<td>35</td>
<td>Guideline</td>
<td>Register &quot;f_out&quot; has not been initialized using an externally controlled asynchronous reset signal</td>
</tr>
<tr>
<td>42</td>
<td>Medium</td>
<td>Avoid using hard coded numeric values such as &quot;1100*&quot; for literals used in assignment expressions</td>
</tr>
<tr>
<td>44</td>
<td>Medium</td>
<td>Avoid using hard coded numeric values such as &quot;1001*&quot; for literals used in assignment expressions</td>
</tr>
<tr>
<td>50</td>
<td>Guideline</td>
<td>Register &quot;g_out&quot; has not been initialized using an externally controlled asynchronous reset signal</td>
</tr>
<tr>
<td>52</td>
<td>Medium</td>
<td>Avoid using hard coded numeric values such as &quot;1100*&quot; for literals used in assignment expressions</td>
</tr>
<tr>
<td>61</td>
<td>High</td>
<td>Next State logic, in FSM &quot;FSM_MEALY_COMP.MEALY&quot;, has no &lt;others&gt; branch for recovery from unused state(s)</td>
</tr>
<tr>
<td>61</td>
<td>Low</td>
<td>Line length exceeds maximum of &quot;93&quot; characters per HDL line</td>
</tr>
</tbody>
</table>

Table 7.8. FSM_MEALY_COMP_entity violation report.
7.4 Running avionics codes

7.4.1 Avionics code (Verified)

When running this code in HDL Design Checker, the design quality is calculated to 49%. The different rules that are violated can be seen in figure 7.3.

The rules are weighted for how many they are. The reset rules are the ones that are violated most of the available rules. For clock rules, most rules are passed.

![Distribution of violations when testing avionics code (verified) in Design Checker.](image)

**Figure 7.3.** Distribution of violations when testing avionics code (verified) in Design Checker.
7.4.2 Avionics code (not verified)

When running this code in HDL Design Checker, the design quality is calculated to 63%. The different rules that are violated can be seen in figure 7.4.

Here can be seen that the reset rules are about 30%, which is the highest category, and that all FSM rules are passed.

![Distribution of violations when testing avionics code (not verified) in Design Checker.](image)

**Figure 7.4.** Distribution of violations when testing avionics code (not verified) in Design Checker.
Chapter 8

Verification Navigator (VN-Check)

In this chapter VN-Check is evaluated.

8.1 General applicability

8.1.1 Running the tool

The different stages in the design-flow are controlled using a “Verification Flow” window. If pressing an icon in the window the user will go to that corresponding function. If using VHDL, first specify which library to use and secondly specify which files to include in the rule check. When this is done the program compiles the library and files and displays any compiling errors that are present.

The next step is to configure VN-Check. Select which rule-sets to use as well as specify the top-level file. Furthermore there are different settings to choose from (e.g. the tool will only compile the project files and not do any rule checking).

Another setting is a limit that no more than 500 violations will be displayed. This will help the user not to be overflowed with violations. To ignore pragmas in the code or not can be chosen here. There are also settings for the output-file, e.g. which name and what catalogue the file will go in, if errors, warnings and/or notes will be displayed, the different rule categories to use and also which rules to exclude from the check.

8.1.2 Rules

The rule has a value that in some cases are predefined and can be chosen from a list and sometimes it is possible write own values, with numbers or regular expressions. A comment can be written for every rule that can explain how to interpret the rule or exceptions from the rule. For every rule there is a link to an explanation of the rule, presenting how it should be used. Furthermore, a search function in
the rule editor is available that will help find the rules.

A rule has different arguments.

- **Current value:** The argument for the rule. Can be predefined values, user-defined integers or strings, or regular expressions.
- **Rule severity:** Severity level of the rule if it is violated.
- **Violation message:** The message that will be displayed, when the rule is violated.
- **Scope of the rule:** Decides where in the hierarchy the rule shall apply.
- **Comments:** More help for the user can be displayed here.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Prefixes and Suffixes</td>
<td>Can be implemented with script.</td>
</tr>
<tr>
<td>2. Lowercase and uppercase</td>
<td>Choose the rule EntityName amongst the naming rules. For the value an example can be: write regular expression: [EntityName_UC] to check that the entity names are in uppercase.</td>
</tr>
<tr>
<td>3. Number of characters in names no more than 25</td>
<td>Choose the rule MaximumIdentifierLength amongst the style rules and the value is set to 25.</td>
</tr>
<tr>
<td>4. Only one VHDL-statement per line</td>
<td>Choose the rule DeclarationsPerLine amongst the style rules and set the value to 1.</td>
</tr>
<tr>
<td>5. Line length with max 93 characters</td>
<td>Use the LineLength rule amongst the style rules. The value is set to 93.</td>
</tr>
<tr>
<td>6. Indentation with 2 spaces</td>
<td>This rule is not fully implemented, but choose ConsistentIndentation amongst the style rules to get some indentation check. This rule will check that the indentation is consistent in the code.</td>
</tr>
<tr>
<td>7. Port-mapping has new lines for each parameter</td>
<td>The rule EntityPortDeclarationsPerLine amongst the style rules will check that there is only one port declaration on each line. Set the value to 1.</td>
</tr>
<tr>
<td>8. Using rising_edge and falling_edge for the clock</td>
<td>Not available.</td>
</tr>
<tr>
<td>9. Either Mealy or Moore state-machines are used in the design</td>
<td>Not available.</td>
</tr>
<tr>
<td>10. FSM shall be created with one sequential process or a combination of two processes, one sequential process that controls the state transitions and one combinational process to control the dataflow</td>
<td>Not available.</td>
</tr>
</tbody>
</table>

Table 8.1. General applicability with VN-Check.

### 8.1.3 Severity levels

There are three available severity levels (note, warning and error). It is possible to write own error messages and if the OpenMORE standard is to be used, a rule weight can be defined for that rule.
8.2 Specific applicability

8.1.4 Result

When using the code checking from the “Verification Flow” window the user can watch the progress in a log window called “Analyze log”. Here the files that are compiled can be seen and if there are any errors, also if there is any FSM in the code can be seen here. If there is, the FSM is seen in a graphical window and how the states are connected to each other. In the log, the progress of the rule checking is shown. For every file in the analysis, the total number of violations is shown. Also the names of the rules that are violated and how many errors and warnings there are in every of the different rule categories. How many rules that have been used and how many of them that passed, this is displayed with both numbers and percentage. This percentage could be some kind of quality assessment, however the percentage for the whole design are not shown.

In the result-file the different violations are shown in more detail than in the log window. There is also a cross reference to a code editor, to see where the violations are in the code. In the result window a filter function is available, to sort out unnecessary data.

8.2 Specific applicability

8.2.1 Rules

The more advanced rules that are implemented in VN-Check are shown in table 8.2. VN-Check will only find the FSMs in the design if they are encoded in the following way.

- The state-transition code has to be in a case (current_state) block.
- If there is only a current_state variable (e.g. no next_state variable exists), then the whole FSM must be in a single, clocked process.
- If there is a next_state variable, the FSM can be in 1 or 2 processes.
- If the FSM is in 2 processes, these should be consecutive (e.g. avoid adding other processes in between.)
- The reset state assignment must be easily identified: avoid multiple reset assignments.
- Verilog recognition is aware of One-hot and One-hot-0 state encodings. VHDL recognition is not, but for VHDL a classical rule is to use an enum-type for the states and give them the desired encoding.
- State naming in the switches of the case statement should be consistent (e.g. all named parameters or all explicit numbers etc.)
Rule | Implementation
--- | ---
11. Buses shall be declared with bits given from MSB to LSB | Choose the rule ArrayDirection amongst the coding rules. Set the value to descending.
12. Latches in the design. | Choose the rule Latches amongst the coding rules. Set the value to disallow. It is also possible to use the rule IfMissingElse in coding rules, and set it to disallow. This rule will disallow that we miss the else statement in IF.
13. Mixed positive- and negative-edge triggered flip-flops. | Not available, only for FSM. For FSM, choose FSMClockType amongst coding rules. Set the value to consistent.
14. Gated clocks in the RTL. | Not available, however, the compiler warns that the design is not synthesizable because of gated clock in the design.
15. Internally generated clocks in the design. | Use the rule InternallyGeneratedClocks in coding rules. Set the value to disallow.
16. Clock domain crossings. | The rules ClockDomainCrossings and HierarchicalCDC can be used here. They are coding rules and will check for CDC in the code.
17. Internally generated resets. | Choose the rule InternallyGeneratedResets amongst coding rules and set the value to disallow.
18. Asynchronous FSMs | Use the rule FSMStyle amongst the coding rules. Set the value to synchronous. This rule will check that all FSMs are synchronous.
19. All outputs from an FSM have to be initiated asynchronously at reset | Not available.
20. All states can transition to a known state | Use the rules TerminalState and UnreachableState, which can be found amongst the coding rules. Set their values to disallow. TerminalState will check for terminal states in the FSMs and disallow this. UnreachableState will look for unreachable states in the FSMs and disallow this.
21. Gray coding | There are no rules available for this, but it may be implemented with script.
22. One hot coding | There are no rules available for this, but it may be implemented with script.

Table 8.2. Specific applicability with VN-Check.

8.2.2 Implement specific rules

It is also possible to create own rule-sets. If own rule-sets are created there are a wide variety of base-rules to choose from in four categories.

- **Coding:** See the implementations in table 8.2.
- **Style:** Here are rules like “LineLength” and “ConsistentIndentation”. The rule “ConstantIndentation” will determine if blocks of similar statements are consistently indented. This can be very useful if a structured and nice looking code is desired.
- **Document:** The rules in this category are focusing on comments in the code. In example 8.1 the “HeaderComment” rule is explained and in example 8.2 the ConstantDeclarationComment rule is explained.
- **Naming:** In this category are rules for naming. In example 8.3 the name rule for the clock is explained.
Example 8.1: HeaderComment in VN-Check

For this rule to work there is a need to distinguish between a header or file comment and a module comment. Comments that are associated with the file should be written with -- +FILEHEADER+ for starting comment and -- -FILEHEADER- for ending.

-- +FILEHEADER+
-- This is a header comment
-- -FILEHEADER-

The HeaderComment can be set to TEMPATE:template.file" if we want a template file to match the header. The following syntax is used with regular expressions:

```
(^--\s*\+FILEHEADER\+\s*)
(^--\s*This is the file comment\s*)
(^--\s*-FILEHEADER-\s*)
```

The () is used for denoting that we use a regular expression. The "^" denotes that "--" must appear at the start of the line. The "\s*" denotes that one or more spaces will match, so

-- +FILEHEADER+
Is the same as

-- +FILEHEADER+

The "\s*" is required because "+" has a meaning to the regular expression engine. Finally, spaces after +FILEHEADER+ will be taken care of by "\s*".

--- Example 8.2: ConstantDeclarationComment in VN-Check ---

-- This constant declaration has a comment
Constant a : bit_vector(1 to 5) := "10102"

If we did not have a comment above the constant, we would get a violation. There are many rules of this kind for the most things in VHDL.
Example 8.3: Naming in VN-Check

If we want the clock signal to be allowed to be named clk1, clk2, clk_interface or clk_CPU, use the rules ClockName and ClockNameIllegal. For the allowed clock names write regular expressions.

(clk\[0-9\]_*)

In the ClockNameIllegal rule write the clock names that are illegal, such as

(*clk*)

And all the other ways we can think of. The rest of the rules in this category are working the same way, with allowed named and illegal names.
8.3 Reliability

After the tool is used on the TEST_PROJECT with the rules according to Appendix B, the results are presented in table 8.3 (The severity levels were not used here).

<table>
<thead>
<tr>
<th>TEST_CODE_entity.vhd</th>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td></td>
<td>ClockDomainCrossings: Clocks 'clk2','clk' cross at net 'out1', Clock Domain Crossing.</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td></td>
<td>HierarchicalClockDomainCrossings: Clocks 'clk2','clk' cross at net 'out1'.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTERNAL_COMP_entity.vhd</th>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td></td>
<td>EntityPortDeclarationsPerLine: Entity port declarations per line (2) exceeds maximum.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GATED_COMP_entity.vhd</th>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td></td>
<td>EntityName: Entity name 'gated_comp' does not match requirement 'GATED_COMP'. Use uppercase for entity.</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td>EntityPortDeclarationsPerLine: Entity port declarations per line (2) exceeds maximum.</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
<td>LineLength: Line length of 96 exceeds maximum.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FSM_MOORE_entity.vhd</th>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14</td>
<td></td>
<td>Latches: Incomplete assignments to 'D1' may cause latch inference.</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td></td>
<td>ArrayDirection: Range found with TO direction, which is inconsistent with requirements.</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td></td>
<td>FSMClockType: FSM clock 'clk' trigger is not consistent with other FSMs in the design.</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td></td>
<td>Asynchronous: FSM reset 'reset' is synchronous. Need asynchronous reset.</td>
</tr>
<tr>
<td></td>
<td>51</td>
<td></td>
<td>TerminalState: State 'st3' is a terminal state.</td>
</tr>
<tr>
<td></td>
<td>62</td>
<td></td>
<td>LineLength: Line length of 111 exceeds maximum.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FSM_MEALY_entity.vhd</th>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td></td>
<td>AsynchronousResetRegisters: Register 'state' has not been asynchronously reset.</td>
</tr>
<tr>
<td></td>
<td>61</td>
<td></td>
<td>LineLength: Line length of 105 exceeds maximum.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CDC_COMP_entity.vhd</th>
<th>Line</th>
<th>Severity</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td></td>
<td>ClockDomainCrossings: Clocks 'clk2','clk' cross at net 'out1', Clock Domain Crossing.</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td></td>
<td>HierarchicalClockDomainCrossings: Clocks 'clk2','clk' cross at net 'out1'.</td>
</tr>
</tbody>
</table>

Table 8.3. Result for TEST_PROJECT in VN-Check.
Chapter 9

Results

In this chapter, the results of the evaluations are summarized and the two tools are compared.

9.1 General applicability

Running the tool

When running Design Checker and VN-Check it is apparent that Design Checker is easier to getting started with than VN-Check, due to a more complicated setup procedure in VN-Check. It should be mentioned that the version of Design Checker that was used was newer than the VN-Check version.

Rules

Most of the more simple rules could be implemented in Design Checker, but only 5 out of 10 for VN-Check. Prefixes and suffixes can be implemented in VN-Check, but using script.

Severity levels

I nice feature in Design Checker is that the severity levels are customizable. They are not in VN-Check.

Results

Design Checker can show the total quality score of the code. VN-Check shows only quality score for each file.
9.2 Specific applicability

Rules
For Design Checker 7 out of 12 advanced rules could be implemented. For VN-Check, also 7 out of 12 could be implemented. VN-Check has CDC check which Design Checker has not.

Implementation of specific rules
Design Checker is using Tcl and VN-Check is using regular expressions for scripting. Tcl is more powerful than regular expressions, so to implement new rules using script, Design Checker is preferred.

9.3 Reliability

To compare the reliability of the tools, only identically implemented rules in both tools will be compared. Only the TEST_PROJECT files that the tools have common rules for are shown.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Design Checker</th>
<th>VN-Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. Number of characters in names no more than 25.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Only one VHDL statement per line.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Line length max 93 characters.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Port-mapping has new lines for each parameter.</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>11. Buses shall be declared with bits given from MSB to LSB.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12. Latches in the design.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15. Internally generated clocks in the design.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17. Internally generated resets.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20. All states can transition to a known state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total score:</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 9.1. INTERNAL_COMP_entity.vhd

<table>
<thead>
<tr>
<th>Rule</th>
<th>Design Checker</th>
<th>VN-Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. Number of characters in names no more than 25.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Only one VHDL statement per line.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Line length max 93 characters.</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>7. Port-mapping has new lines for each parameter.</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>11. Buses shall be declared with bits given from MSB to LSB.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12. Latches in the design.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15. Internally generated clocks in the design.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17. Internally generated resets.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20. All states can transition to a known state.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total score:</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 9.2. GATED_COMP_entity.vhd
9.3 Reliability

<table>
<thead>
<tr>
<th>Rule</th>
<th>Design Checker</th>
<th>VN-Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. Number of characters in names no more than 25.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Only one VHDL statement per line.</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>5. Line length max 93 characters.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Port-mapping has new lines for each parameter.</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>11. Buses shall be declared with bits given from MSB to LSB.</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>12. Latches in the design.</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>15. Internally generated clocks in the design.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17. Internally generated resets.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20. All states can transition to a known state.</td>
<td>0/1</td>
<td>1/1</td>
</tr>
<tr>
<td>Total score:</td>
<td>75%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 9.3. FSM_MOORE_entity.vhd

In table 9.3 it can be seen that Design Checker misses that a state does not have anywhere to go, but VN-Check discovers this.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Design Checker</th>
<th>VN-Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. Number of characters in names no more than 25.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Only one VHDL statement per line.</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>5. Line length max 93 characters.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Port-mapping has new lines for each parameter.</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>11. Buses shall be declared with bits given from MSB to LSB.</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>12. Latches in the design.</td>
<td>1/1</td>
<td>0/1</td>
</tr>
<tr>
<td>15. Internally generated clocks in the design.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17. Internally generated resets.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20. All states can transition to a known state.</td>
<td>0/1</td>
<td>1/1</td>
</tr>
<tr>
<td>Total score:</td>
<td>100%</td>
<td>50%</td>
</tr>
</tbody>
</table>

Table 9.4. FSM_MEALY_entity.vhd

In table 9.4 it can be seen that VN-Check misses that there is no “when others” in the code. Design Checker discovers this.
9.4 Summary

9.4.1 Design Checker and its advantages and disadvantages.

+ Already predefined rules libraries with rules from RMM
+ Different kinds of severity levels that the user can customize
+ Possibility to organize rules into sets, and organize the sets into policies
+ Reporting of total quality scoring
+ Exporting of result-files
+ FSM Graphics
+ Tcl scripting support

- It is not possible to make entirely new rules in the tool. There is a need to write scripts that will be run from the log window. Another solution is to let Mentor Graphics make the rules, which may be costly.
- No CDC check

9.4.2 VN-Check and its advantages and disadvantages.

+ Already predefined rules libraries with rules from RMM
+ Creation of new rules databases from the supplied rules
+ CDC check
+ Large number of documentation rules
+ Exporting of result-files
+ FSM graphics

- It is not possible to create entirely new rules
- No total quality score
- Not so easy to get started
Chapter 10

Conclusions

10.1 Motivation of achieved goals

The first goal was to strengthen Saab Avitronics knowledge in adaptable rule checking tools for HDL. This goal was accomplished in chapter 2.1 (describing what a rule checking tool is), chapter 3 (Principles of a rule checking tool for HDL) and chapter 5 (a market analysis of available tools for HDL).

The second goal was to suggest how Saab Avitronics should implement their own rules and which tool(s) is/are the most suitable. This goal was achieved in chapter 6 (Evaluation of rule checkers), 7 (Evaluation of Design Checker), 8 (Evaluation of VN-Check) and 9 (Results).

The last goal was to analyze some of the safety-critical problems that can arise when violating the rules. This goal was accomplished in chapter 4 (Design issues in HDL for safety-critical systems).

10.2 Discussion about the result

Both of the tools performed ok in the reliability test. Some things were not found by the tools, but most of them were. However, the test with the testcode was not that helpful, to say anything about the reliability of the tools. It was too many rules that the tools did not have in common and the code was not that extensive. To really be able to say something about the reliability, a more extensive test should be made. This would of course take a lot of work and time, which was not available for the scope of this thesis. Despite of this limitation, the performed reliability test gives some kind of indication on what the tools are able to find in the code.

To recommend one of the tools after the evaluations, the author recommends Design Checker over VN-Check. It was easier to analyze the code and to analyze the results. Also more of Saab Avitronics rules could be implemented in Design Checker.

Another solution is a combination of two or more rule checking tools. The
tools have different strengths and weaknesses and could complement each other. Design Checker has its strengths in providing most of the coding rules that are needed and VN-Check has strengths in documentation and style.

The author believes that if an inexperienced programmer is coding the safety-critical systems, the faults are more likely to occur, than if an experienced does it. So the introduction of a static rule checker would help the inexperienced programmers. It is also not possible to guarantee that the static rule checking tool will find every possible hazards in the code. Someone will still have to check the code for certain issues (for safety-critical projects, this is still a requirement).

However, the most basic faults and problems in the layout of the code can be found by the tool. If these faults are taken care of first, then the more advanced faults are easier to discover and the validator can concentrate on more advanced faults and therefore save time. It seems strange that the verified aviation code has lower quality than the one that is not verified. The main reason for this is that the codes are completely different. The best would be to have the same code that could be checked before and after verification. However, this was not possible at the time, maybe this could be tested in the future. Another reason is that several rules may be intentionally broken by the designer. This kind of “valid broken rules” is not taken care of in this thesis.

As can be seen from the previous chapters it is easy to implement the rules and most of the rules are already defined. The only thing that is needed to do is to parameterize them to suit our needs. Only a few rules are implemented using scripts, and this is how we do it if we do not find any rules to parameterize.

Finally it is concluded that the introduction of static rule checking tools will help to lower the iterations in the FPGA and HDL design flow. They will also warn of dangerous constructs. Moreover will they help to write designs in a reusable way, following a certain reuse methodology, so designs can be used in other projects. However, it will not be possible to fully automate rule checking for safety-critical systems, because of the high requirements on reliability.
Chapter 11

Future work

In the future, all of the rules from Saab should be implemented. For the rules that cannot be implemented with only parameterization of the already existing rules, there is a need to write scripts for certain rules. To implement the scripts into the design flow, there is a need to run the rule checking tools from batch mode and disregard the GUI. The GUI can be good if wanting to learn how the tools work and to organize and modify the parameterized rules. A script can run the rule checking tools and produce result files that can be examine. To implement scripted rules into the result files there is a need to alter the result file, and recalculate the quality percentages and rule information. Otherwise the scripted rules have to be run separately and have a separate result file and quality report.

The best would be to have a rule checking tool that had a rule editor, were user made rules can be written and add them to the tool. Then everything could be run from the GUI mode, which can be easier to handle than the scripting environment. The only tool so far that has the ability to write own rules in an editor is LEDA. This tool was not evaluated any further in this thesis, but it could be interesting to see the capabilities of the rule editor. Both Mentor Graphics and TransEDA offer the solution that they can create rules if we specify them. However, it is not wanted that any third party is getting knowledge about exactly what kind of rules Saab is using, especially not the ones concerning safety-critical systems. Leda is supplying their own language to write rules in the rule editor. The restrictions to use templates can maybe be too restrictive in some cases. The language for the rule editor can be difficult to understand, so maybe there is a need to make contact with a third party anyway.

This thesis was only looked at commercial tools. It can be beneficial to look at open source tools as well. They would surely be easier to modify and to write completely own rules.
Bibliography


69
Appendix A

The rules that are implemented into Design Checker

<table>
<thead>
<tr>
<th>Rule category</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format and style</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guideline (File Name)</td>
<td>Lowercase</td>
<td>Checks that the filename is in lowercase.</td>
</tr>
<tr>
<td>Rule3 (Lowercase)</td>
<td>Lowercase</td>
<td>Checks that signals, variables and ports are in lowercase.</td>
</tr>
<tr>
<td>Rule4 (Uppercase)</td>
<td>Uppercase</td>
<td>Checks that components, architectures, entities, instances, constants are in uppercase.</td>
</tr>
<tr>
<td>Rule6 (Length)</td>
<td>Max length 25</td>
<td>Checks that no names are longer than 25 characters.</td>
</tr>
<tr>
<td>Rule10 (Descending bus order)</td>
<td>Descending</td>
<td>Checks that the order of the bits is descending for buses.</td>
</tr>
<tr>
<td>Rule11 (Statement Style)</td>
<td></td>
<td>Checks that there are only one statement per line.</td>
</tr>
<tr>
<td>Rule12 (Length)</td>
<td>Max length 93</td>
<td>Checks that the rows are not longer than 93 characters.</td>
</tr>
</tbody>
</table>

Table A.1. Rules implemented into Design Checker
### Rule category | Value | Explanation
--- | --- | ---
**Prefixes**
Label each generic signal $g\_name$ | Regular expression: $g\_*$ | Checks that each generic signal has the prefix $g\_$.  
Label each instance $U\#\_name$ | Regular expression: $U\#\_*$ | Checks that each instance has the prefix $U\#\_$.  
Label each sequential process $seq\_name$ | Regular expression: $seq\_*$ | Checks that each sequential process has the prefix $seq\_$.  
Label each variable $v\_name$ | Regular expression: $v\_*$ | Checks that each variable have the prefix $v\_$.  
**Suffixes**
Label each configuration $name\_cfg$ | Regular expression: $\_cfg$ | Checks that each configuration has the suffix $\_cfg$.  
Label each FSM current state $name\_current\_state$ | Regular expression: $\_current\_state$ | Checks that each FSM current state has the suffix $\_current\_state$.  
Label each FSM next state $name\_next\_state$ | Regular expression: $\_next\_state$ | Checks that each FSM next state has the suffix $\_next\_state$.  
Label each package $name\_pkg$ | Regular expression: $\_pkg$ | Checks that each package has the suffix $\_pkg$.  
Label each process $name\_proc$ | Regular expression: $\_proc$ | Checks that each process has the suffix $\_proc$.  

Table A.2. Rules implemented into Design Checker
<table>
<thead>
<tr>
<th>Rule category</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clock</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guideline (Edge detection)</td>
<td>VHDL edge-function</td>
<td>Checks that <code>rising_edge</code> and <code>falling_edge</code> are used for clock edge instead of clock event.</td>
</tr>
<tr>
<td>Rule25 (Avoid internally generated clocks)</td>
<td>Disallow</td>
<td>Checks for internally generated clocks.</td>
</tr>
<tr>
<td>Rule26 (Avoid Gated clocks in the RTL)</td>
<td>Disallow</td>
<td>Checks for gated clocks in the design.</td>
</tr>
<tr>
<td>Rule27 (Isolate gated and internally generated clocks)</td>
<td>Allow if isolated at top-level</td>
<td>Checks that gated clocks and internally generated clocks are isolated at the top level.</td>
</tr>
<tr>
<td><strong>Reset</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule31 (Reset logic function)</td>
<td></td>
<td>Checks that the reset signals are only used to reset the flip-flops in the design.</td>
</tr>
<tr>
<td>Rule32 (Isolate conditional resets)</td>
<td></td>
<td>Checks that conditional resets are isolated into separate modules.</td>
</tr>
<tr>
<td>Rule33 (Avoid internally generated resets)</td>
<td>Disallow</td>
<td>Checks for internally generated resets.</td>
</tr>
<tr>
<td><strong>State-Holding elements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guideline (Inferred registers with asynchronous reset)</td>
<td></td>
<td>Checks that all registers has a global asynchronous reset.</td>
</tr>
<tr>
<td>Rule21 (Avoid latches)</td>
<td></td>
<td>Checks for latches in the design.</td>
</tr>
<tr>
<td><strong>VHDL Element</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSM coding style</td>
<td>Moore</td>
<td>Checks that the FSMs is described in a consistent style.</td>
</tr>
<tr>
<td>FSM state encoding style</td>
<td></td>
<td>Checks for redundant states.</td>
</tr>
<tr>
<td>FSM transition</td>
<td></td>
<td>Checks that all states can transition to a recovery state and a known state.</td>
</tr>
<tr>
<td>Guideline (Sensitive list)</td>
<td></td>
<td>Checks that only signals that are needed in the sensitivity list are in it.</td>
</tr>
<tr>
<td>Rule34 (Do not use other HDL reserved words)</td>
<td></td>
<td>Checks that Verilog reserved words are not used in the VHDL design.</td>
</tr>
<tr>
<td>Rule42 (Hard coded numeric values)</td>
<td></td>
<td>Checks that we are not using hard coded values in ranges and operands.</td>
</tr>
<tr>
<td><strong>Signal</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule35 (Use std_logic_1164 types only)</td>
<td>std_logic_1164</td>
<td>Checks that top level ports use <code>std_logic_1164</code>.</td>
</tr>
</tbody>
</table>

Table A.3. Rules implemented into Design Checker
Appendix B

The rules that are implemented into VN-Check
<table>
<thead>
<tr>
<th>Rule category</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ArrayDirection</td>
<td>Descending</td>
<td>Checks that the bits in the bus are declared with descending order.</td>
</tr>
<tr>
<td>AsynchronousResetRegisters</td>
<td>Required</td>
<td>Checks that all registers are asynchronously reset.</td>
</tr>
<tr>
<td>ClockDomainCrossings</td>
<td>Disallow</td>
<td>Checks for clock domain crossings.</td>
</tr>
<tr>
<td>FSMClockType</td>
<td>Consistent</td>
<td>Checks that the clock edge trigger is consistent.</td>
</tr>
<tr>
<td>FSMReset</td>
<td>Required</td>
<td>Checks that the FSM has a reset.</td>
</tr>
<tr>
<td>FSMResetType</td>
<td>Asynchronous</td>
<td>Checks that the FSM is asynchronously reset.</td>
</tr>
<tr>
<td>FSMStyle</td>
<td>Synchronous</td>
<td>Checks that the FSM is synchronous.</td>
</tr>
<tr>
<td>HierarchicalCDC</td>
<td>Disallow</td>
<td>Checks for clock domain crossings.</td>
</tr>
<tr>
<td>IfMissingElse</td>
<td>Disallow</td>
<td>Checks for missing else statements in if statements.</td>
</tr>
<tr>
<td>InternallyGeneratedClocks</td>
<td>Disallow</td>
<td>Checks for internally generated clocks.</td>
</tr>
<tr>
<td>InternallyGeneratedResets</td>
<td>Disallow</td>
<td>Checks for internally generated resets.</td>
</tr>
<tr>
<td>Latches</td>
<td>Disallow</td>
<td>Checks for latches in the design.</td>
</tr>
<tr>
<td>TerminalState</td>
<td>Disallow</td>
<td>Checks if the FSM has a terminal state that would not lead anywhere.</td>
</tr>
<tr>
<td>UnreachableState</td>
<td>Disallow</td>
<td>Checks for unreachable states in the FSM.</td>
</tr>
<tr>
<td>Style</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConsistentIndentation</td>
<td>Required</td>
<td>Checks that the indentation is consistent.</td>
</tr>
<tr>
<td>DeclarationsPerLine</td>
<td>1</td>
<td>Checks that there is only one declaration per line.</td>
</tr>
<tr>
<td>EntityPortDeclarationsPerLine</td>
<td>1</td>
<td>Checks that there is only one port declaration per line.</td>
</tr>
<tr>
<td>LineLength</td>
<td>93</td>
<td>Checks that the maximum row length is 93 characters.</td>
</tr>
<tr>
<td>MaximumIdentifierLength</td>
<td>25</td>
<td>Checks that names are not longer than 25 characters.</td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naming</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EntityName</td>
<td>Regular expression: [EntityName_{UC}]</td>
<td>Checks that entities are in uppercase.</td>
</tr>
</tbody>
</table>

Table B.1. Rules implemented into VN-Check
Appendix C

Testcode

TEST_CODE_entity.vhd

1. -- Filename: TEST_CODE_entity.vhd --Use lowercase for entity
2. -- Created: 2008-02-21
3. -- Author: Mikael Lord
4. -- Description: Top-level port-mapping for TEST_PROJECT.
5.
6. LIBRARY ieee;
7. USE ieee.std_logic_1164.all;
8. USE ieee.std_logic_arith.all;
9.
10.
11. ENTITY TEST_CODE IS
12. port(clk : in std_logic;
13. reset : in std_logic;
14. enable : in std_logic;
15. x : in std_logic;
16. y : in std_logic;
17. z : in std_logic;
18. in1 : in std_logic;
19. clk2 : in std_logic;
20. out1 : out std_logic;
21. out2 : out std_logic;
22. g_out : out std_logic;
23. f_out : out std_logic_vector(3 downto 0);
24. d_out : out std_logic_vector(3 downto 0));
25. END ENTITY TEST_CODE;
26.
27. architecture TOP_LEVEL of TEST_CODE is
28.
29. signal clk_c : std_logic;
30. signal D1 : std_logic; --Use lowercase for signal
31.
32. begin
33. U1 : entity work.gated_comp rtl port map(clk,enable,clk_c); --Use U#_* for instances
34. U2 : entity work.INTERNAL_COMP(BEHAVE) port map(clk_c,z,out2); --Use U#_* for instances
35. U3 : entity work.CDC_COMP(CDC) port map(clk,clk2,in1,out1); --Use U#_* for instances
36. U4 : entity work.FSM_MOORE_COMP(MOORE) port map(clk,reset,x,D1,d_out); --Use U#_* for instances
37. U5 : entity work.FSM_MEALY_COMP(MEALY) port map(clk2,reset,y,D1,g_out,f_out); --Use U#_* for instances
38.
39. end;
GATED_COMP_entity.vhd

1. -- Filename: GATED_COMP_entity.vhd --Use lowercase for entity
2. -- Created: 2008-02-21
3. -- Author: Mikael Lord
4. -- Description: Gated clock for TEST_PROJECT.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY gated_comp IS
  port(clk, enable: in std_logic; -- Only one port mapping declaration per line.
  clk_c: out std_logic);
END ENTITY gated_comp;

architecture rtl of gated_comp is
begin
  process(clk,enable)
  begin
    if rising_edge(clk) and enable='1' then -- Gated clock
      clk_c<='1';
    end if;
    clk_c<='0';
  end process;
end RTL;

INTERNAL_COMP_entity.vhd

1. -- Filename: INTERNAL_COMP_entity.vhd --Use lowercase for entity
2. -- Created: 2008-02-21
3. -- Author: Mikael Lord
4. -- Description: Internal clock for TEST_PROJECT.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY INTERNAL_COMP IS
  port(clk_c, z: in std_logic; -- Multiple declarations for port mapping
  out2: out std_logic);
END ENTITY INTERNAL_COMP;

architecture BEHAVE of INTERNAL_COMP is
begin
  process(clk_c) -- Using internally generated clock
  begin
    if rising_edge(clk_c) then
      out2<='0';
    end if;
    out2<='1';
  end process;
end BEHAVE;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY FSM_MOORE_COMP IS
  port(clk : in std_logic;
       reset : in std_logic;
       x : in std_logic;
       D1 : out std_logic;
       d_out : out std_logic_vector(0 to 3)); -- Buses shall be declared consistent.
END ENTITY FSM_MOORE_COMP;

ARCHITECTURE MOORE of FSM_MOORE_COMP IS
  type state_type is (st0,st1,st2,st3);
  signal state : state_type; -- *_current_state, *_next_state
  begin
    process(clk) -- No asynchronous reset.
    begin
      if reset='1' then
        state<=st0;
      elsif falling_edge(clk) then -- Mixing of rising_edge and falling_edge.
        case state is
        when st0 => if x='1' then
          state<=st1;
          else
          state<=st3;
          end if;
          when st1 => if x='1' then
            state<=st2;
            else
            state<=st0;
            end if;
            when st2 => if x='1' then
              state<=st3;
              else
              state<=st1;
              end if;
              when st3 => D1<='0'; -- All states must transition to a known state.
        end case;
      end if;
    end process;

    process(state)
    begin
      case state is
      when st0 => if x='1' then
        state<=st1;
        else
        state<=st3;
        end if;
        when st1 => if x='1' then
          state<=st2;
          else
          state<=st0;
          end if;
          when st2 => if x='1' then
            state<=st3;
            else
            state<=st1;
            end if;
            when st3 => D1<='0'; -- All states must transition to a known state.
      when others =>state<=st0;
      end case;
    end if;
  end process;

process(state)
begin
  case state is
  when st0=>d_out<="0101";
  when st1=>d_out<="0110";
  when st2=>D1<='1'; -- D1 should be assigned for every state, Latch inferred.
  when st3=>d_out<="01101";
  end case;
end process;
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_arith.all;

Entity FSM_MEALY_COMP is
  Port(clk2 : in std_logic;
       reset : in std_logic;
       y : in std_logic;
       D1 : in std_logic;
       g_out : out std_logic;
       f_out : out std_logic_vector(3 downto 0));
End Entity FSM_MEALY_COMP;

Architecture MEALY of FSM_MEALY_COMP is
  Type state_type is (S0,S1,S2,S3);
  Signal state : state_type; --_current_state, _next_state
  Signal y : in std_logic;
  Signal D1 : in std_logic; --Use lowercase for signal
  Signal f_out : out std_logic;
  Signal g_out : out std_logic_vector(3 downto 0); -- CDC, D1 is clocked by clk.
Begin
  -- Mealy FSM
  Process(clk2, reset)
  Begin
    If reset='1' then
      state<=S0;
    ElseIf rising_edge(clk2) then
      Case state is
        When S0 => if y='1' then
          state <= S1;
          f_out <="1001";
        Else
          f_out <="0000";
        End if;
      When S1 => if y='0' then
        state <= S2;
        f_out <="1100";
      Else
        f_out <="1001";
      End if;
    When S2 => if y='1' then
      state <= S3;
      f_out <="1111";
    Else
      f_out <="1100";
    End when;
  End Case;
    End if;
End Process;
End;
when S3 => if y='0' then
    state <= S0;
    f_out<="0000";
  else
    f_out<="1111";
  end if;
end case; -- When others statement is missing, Latch inferred.
end if;
end process;
end;

CDC_CMP_entity.vhd

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY CDC_CMP IS
  port(clk : in std_logic;
       clk2 : in std_logic;
       in1 : in std_logic;
       out1 : out std_logic);
END ENTITY CDC_CMP;

architecture CDC of CDC_CMP is
  signal sig1 : std_logic;
  begin
  process(clk)
  begin
    if rising_edge(clk) then
      sig1<=in1;
    end if;
  end process;

  process(clk2)
  begin
    if rising_edge(clk2) then
      out1<=sig1; -- Clock domain crossing.
    end if;
  end process;
end CDC;
Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/

© Mikael Lord