Digital Hardware Projects: A New Tool for Automated Complexity Analysis

Benjamin Menhorn, Lukas Brix and Frank Slomka
Ulm University
Institute of Embedded Systems/Real-Time Systems
Ulm, Germany
benjamin.menhorn@uni-ulm.de, lukas.brix@outlook.com, frank.slomka@uni-ulm.de

***Abstract*—This papers introduces a new tool for an automated complexity analysis of digital hardware projects. This analytical tool parses hardware implementations written in VHDL and calculates their complexity. The design entropy is used as a complexity metric. This allows to obtain complexity figures of hardware projects, useful for developers as well as project managers.**

I. Introduction

In 1965 Gordon Moore published an article [1], in which he predicted an annual doubling of the amount of transistors which can be placed inexpensively on integrated circuits for the next 10 years. The interval in which the amount of transistors doubles was later corrected to 18 to 24 months, depending on the source [2] [3] [4]. Moore's observation is known as *Moore's Law*. Figure 1 charts the trend of integrating transistors on a single chip over the past two decades. As for today, Moore's Law is still valid and it may also be for the next decades [5].

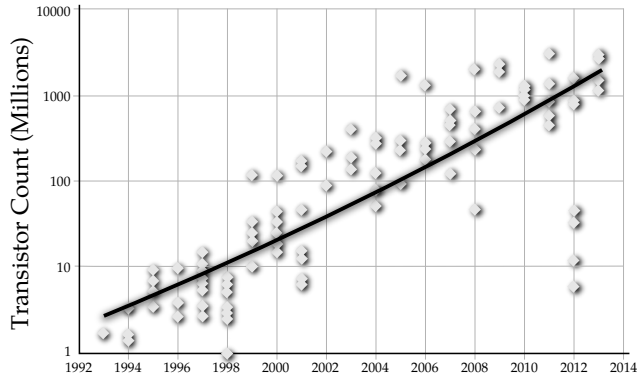

Figure 1. Chip complexity development (from [6])

The significance of this development for planning and implementing hardware designs is enormous. With a constant chip area and an increasing number of components, complexities of chips are growing due to more and more are components placed on the same chip area. And this development is not linear but exponential as expressed in Moore's Law.

In most classical engineering disciplines it is key to have well-defined models and methods to evaluate, control and manage projects. In order to do so, it is necessary to know the project size. This allows to give statements about resources and optimizations during the development process as well as allows to compare different implementations and projects [7] [8]. For both, project managers and engineers themselves, it is important to know about the hardware complexity of a project. During the project, it is useful to always carry a current complexity estimate. Finally, a follow-up assessment is important in order to assess and compare actual results. The use of hardware description languages provides new ways to analyze hardware projects. With help of a suitable metric, complexity of hardware projects can be expressed in figures. For an expressive and profitable applicability of a complexity measure, the evaluation has to be automated, especially for larger projects.

In this paper we introduce a new tool, which calculates complexity figures for hardware designs. We focused on designs which are implemented in VHDL. Our tool itself is written in Java and is platform-independent. The design entropy concept [9] serves us as complexity measure. This measure is based on abstract states.

In the following, the main aspects of the development and implementation of our analytical tool are documented. The key contributions of this paper are:

- The introduction of our complexity analyzing tool for VHDL implementations.
- Show how this tool development is supported by a compiler-compiler.
- Illustrate how our tools works.
- Show first results of or tool.

This paper is organized as follows: The next sections introduces our implementation approach with a compiler, a grammar description of VHDL and a compiler-compiler. Section three explains our methodology including the used complexity measure and the integration of the measurement within the compiler-compiler. Section four shows, how our tool calculates the complexity of VHDL designs. This paper closes with a conclusion and future work.

II. Implementation Approach

In this section, we describe how we used a compiler, the **E**xtended **B**ackus-**N**aur **F**orm (EBNF) and a compiler-compiler to parse VHDL-code.

A. Compiler

A compiler is a software program that translates source code written in a particular language into another language.

978-1-4799-0658-1/13/$31.00

Typically, a compiler is used to translate programs written in a high level language (such as Java or C) into assembler or machine languages, which can then be executed by a computer. Some compilers require preprocessors that combine source codes from individual modules and if necessary support macro substitution and conditional compilation.

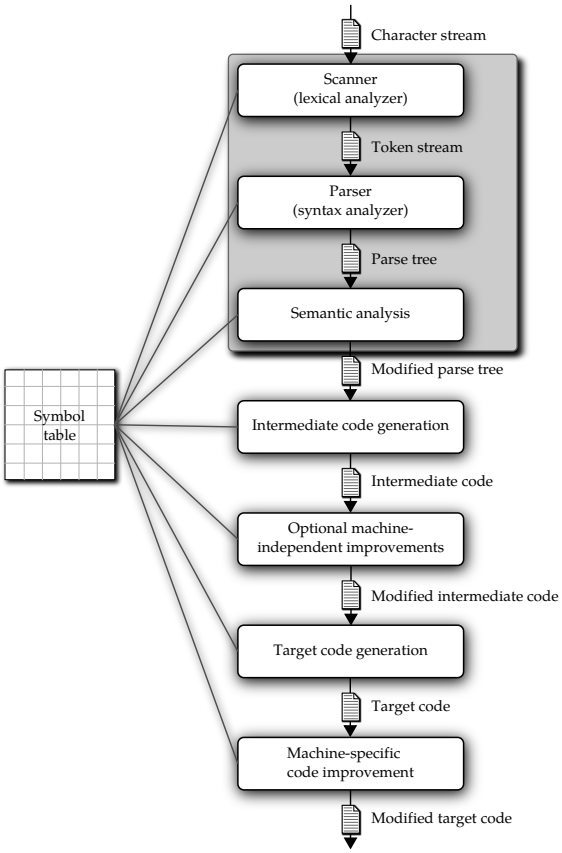

Figure 2. Phases of compilation (based on [10])

Figure 2 illustrated the different stages during a compilation process. In the various stages the code is passed on to the next stage in a specific representation form. Each phase has access to a common symbols table. The symbol table holds information for each identifier of the program regarding its appearance and declaration in the source code. The structure of the compiler can be roughly divided into two parts. In the first part, the source code is analyzed. This part is indicated by the gray background in the figure. In the second part of a compiler, code is synthesized and translated into its target language. In this work, we are only interested in the analytical part. This part allows us to parse source code and analyze it for its complexity. The complexity calculation will be placed directly into the analysis phase of the compiler.

B. EBNF

Source code can be of any length and have infinite combinations of allowed language constructs. Therefore, a representation of the underlying language is needed. It has to represent the complete set of words over a given alphabet of a language without being infinitely large. Grammars are used to derive a complete language with aid of a starting symbol and a finite number of substitution rules. The EBNF is a meta-language which allows grammar definitions. In our case, hardware descriptions/implementations are given in VHDL. The official IEEE standard [11] describes VHDL in EBNF. This grammar will be used as basis for our compiler and complexity analysis. The EBNF description from the IEEE standard has to be implemented in such a way that it can serve as input grammar for a compiler-compiler.

C. ANTLR

To make development of our analysis tool more efficient and to avoid sources of errors in programming, we base our tool generation on a so-called *compiler-compiler*. Such compiler-compilers generate parsers, interpreters or compilers from a formal language description. As mentioned above, we only need the analytical part of a compiler. Thus, we could use almost any compiler-compiler to develop our tool. Here, our choice fell on ANTLR[1]. ANTLR stands for **AN**other **T**ool for **L**anguage **R**ecognition. It is a parser generator and published under the BSD License. ANTLR is written in Java, but it supports a variety of other programming languages.

We decided to use ANTLR for several reasons: first of all, ANTLR is very well documented. Secondly, the generated compiler code remains readable and clearly arranged. Also, ANTLR allows to attach actions to grammar elements directly with in the grammar source. These actions, written in Java, are then embedded into the source code of the parser at the appropriated points. We use this actions for our complexity calculations. Last, there are several plugins for the Eclipse development environment and a IDE, called *ANTLR-Works*, which make development more efficient, especially grammar rule definitions.

III. Methodology

In this section, we give a brief introduction on the used complexity measurement. We also describe, how we integrated the complexity measure into our compiler.

A. Complexity measure

We use the design entropy concept from [9] as our complexity measure. The design entropy bases on Shannon's information entropy. Shannon's information theory [12], formulated by Claude Elwood Shannon, provides mathematical statements about transmitted information. The basic idea behind the design entropy concept is the transmission of information between components. These components can be sources and drains of information. Connections, usually realized by wires in a hardware design, are channels, which transmit information. Information are symbols from a pool of available symbols transmitted over a channel. As a basic model, available symbols can be modeled as "high" and "low" signals. But the available symbols can also be more complex.

With this understanding of transmission of information, a VHDL assignment `a <= b` could be identified as: the information *(=signal level)* from component *(=source/sender)* `b` is transmitted *(=assigned)* to component *(=drain/receiver)* `a`. This transmission of information is illustrated in figure 3.

As the compiler has the ability to go through the VHDL code and identify such assignments, we can calculate the

[1] http://www.antlr.org/

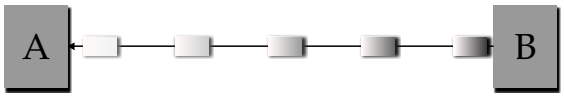

Figure 3. Transmission of information between two components

complexity of a VHDL implementation by parsing the implementation code. Therefore, the EBNF grammar has to be annotated at the corresponding expressions, where any kind of information interchange happens.

This complexity measurement, the design entropy concept, bases its calculations on states, which are a pool of available symbols. "Complexity can be considered to be a measure of a system's disorder which is a property of a system's state. Complexity varies with changes made at the amount of possible states of a system" [13]. Therefore, states allow to measure project size by the measurement of its entropy.

$$H = -K \sum_{\alpha=1}^{N} p_\alpha \log p_\alpha \tag{1}$$

In information theory, Shannon represented the entropy by the theorem given in equation (1). p_α is the likelihood that a system is in cell α of its phase space. This basic formula can be rewritten to the design entropy equations as shown in [9]. This leads to definition 1 for the general behavior entropy from [9].

Definition 1 (Behavior Entropy)**.** *Let c be a component with inputs (component sources) $n_i(c)$, $i = \{1, \ldots, n(c)\}$ and outputs (component drains) $m_j(c)$, $j = \{1, \ldots, m(c)\}$, where $n(c)$ is the amount of inputs of c and $m(c)$ the amount of outputs of c. Let $z(n_i(c))$, $i = \{1, \ldots, n(c)\}$ be the amount of possible states of the inputs $n_1(c) \ldots n_{n(c)}(c)$ and $z(m_j(c))$, $j = \{1, \ldots, m(c)\}$ be the amounts of possible states of the outputs $m_1(c) \ldots m_{m(c)}(c)$. Then the behavior entropy $H_B(c) \in \mathbb{R}_0^+$ of component c is defined as:*

$$H_B(c) = log \left(\prod_{i=1}^{n(c)} z(n_i(c)) \cdot \prod_{j=1}^{m(c)} z(m_j(c)) \right) \tag{2}$$

As most projects, hardware projects are also split into components. These components allow reusability. Therefore the complexity measure as well as our tool need to allow for component based approaches. But the complexity for an instance of a component is not the same as the complexity to build such a component. Therefore, there is a second complexity calculation provided, the structure entropy.

The behavior entropy, given in the definition before, gives statements about the usage complexity of a component. Or in other words the complexity to instance a component. One can also speak of a black box or outer look on a component. The structure entropy, given in the following definition, allows for the actual implementation of a component. Or in other words the complexity to actually build this component. The structure entropy is similar to a white box or an inner view on a component. The structure entropy is given in definition 2 from [9].

Definition 2 (Structure Entropy)**.** *Let c be a component with instances c_b and implemented sub components c_s. Then the structure entropy $H_S(c) \in \mathbb{R}_0^+$ for component c is given by the sum of all behavior entropies of all instances c_b and the structure entropy of all implemented sub-components c_s:*

$$H_S(c) = \sum_{i \in c_b} H_B(i) + \sum_{j \in c_s} H_S(j) \tag{3}$$

With the two different complexity measures, the tool has to distinguish between components, that are implemented within the project and components only used in the project. If a component was implemented within the project, its complexity for the development has to be added to the complexity of the whole project. If the component is just instanced in the project, only the complexity to use to component adds up to a project's complexity.

B. Integration

Apart from the compiler itself, an executable program is required which makes the call. This program consists of a main-method and a calculate-method. A global memory holds the results of already analyzed components. The main-method includes a query of those components, that have to be analyzed as well as their results. These results are determined by calling the calculate-method. In VHDL, the name of a component also serves as file name for this files. As listing 1 shows, the name of a component (*id*) is passed to a recursive call of the main-method (*Run.calculate*) where the file name equals the name of this component (*id.text*). The listing also shows, how actions can be directly embedded into grammar files in ANTLR, such as the recursive call here. These actions are within the braces (curly brackets).

Before results are calculated, the calculation-method checks, whether the desired result has already been calculated and is included in the cache. For the calculation, the source code of a component is read line-by-line and packed into a string. At this step, all comments are removed and all letters are converted to lowercase. This allows a more efficient analysis and doesn't effect the results due to VHDL does not distinguish between uppercase and lowercase. Then, this string is used to call the lexer, the parser, and finally the calculated complexities are returned. We always consider the input code to our tool as grammatically correct and we do not check for error.

```
1  component_declaration:
2    'component' id = IDENT ('is')?
3      ( generic_clause )?
4      ( port_clause )?
5    'end' 'component' ( IDENT )? ';'
6    {
7      //recursive call for a sub-component
8      ResultTriple rt = Run.calculate($id.text);
9      //save to complexity figures
10     $design_file::memory.put($id.text, rt);
11   }
12 ;
```

Listing 1. Component declaration with an action

IV. Experimental Results

Figure 4 shows the terminal in-/output of our tool. After the start of the tool, it asks for the name of the top-level-entity. As a small example we analyze a ripple-carry-adder. This

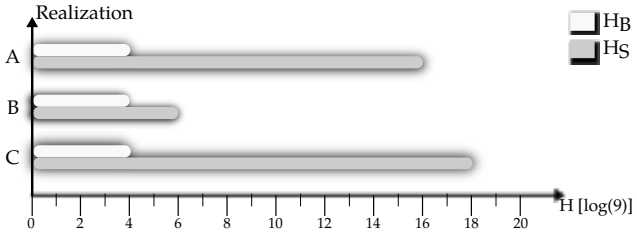

Figure 6. Entropy of three half-adder realizations

```
$java source.Run
Name of the program to analyze (w/o .vhd):
ripple_carry_adder

Top-Level-Entity: ripple_carry_adder
=====================================

--> Component ripple_carry_adder found. Started analysis...

--> Component full_adder found. Started analysis...

--> Component half_adder found. Started analysis...

<-- Component half_adder
    Behavior Entropy:  4 * log(9)
    Structure Entorpy: 24 * log(9)

<-- Component full_adder
    Behavior Entropy:  5 * log(9)
    Structure Entorpy: 35 * log(9)

<-- Component ripple_carry_adder
    Behavior Entropy:  14 * log(9)
    Structure Entorpy: 55 * log(9)

=====================================
Behavior Entropy for the top-level-entity ripple_carry_adder  : 14 * log(9)
Structure Entropy for the top-level-entity ripple_carry_adder : 55 * log(9)

$
```

Figure 4. Tool in the terminal

ripple-carry-adder consists of full-adders, as figure 5 illustrates. As the full-adders were also implemented, the tool needs to analyze them, too. The full-adders consist of two half-adders and one OR-gate. These half-adders are build by NAND-gates. The OR-Gate used in the full-adder and the NAND-gates used in the half-adders are basic gates. Therefore, those gates are not considered as implemented components. For the complexity calculation only their behavior entropy has to be calculated. As the output of the tool in figure 4 shows, the single components are hierarchically analyzed. After the complexity has been calculated for all components, the complexity for the top level entity is calculated and print out. The tool can also analyze more complex implementations with several sub-components.

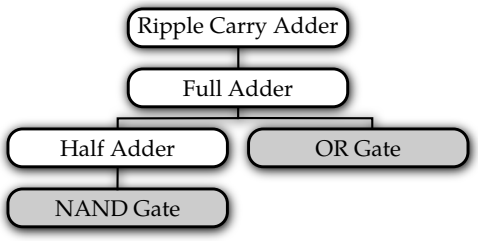

Figure 5. Structural design hierarchy of a ripple carry adder

Complexity figures of different designs allow to compare these designs with each other. Especially, different designs for the same task can be directly compared by numbers. As an example, the following box shows the complexity calculation results for three different half-adder designs:

```
Behavior Entropy for the top-level-entity half_adder_a  : 4 * log(9)
Structure Entropy for the top-level-entity half_adder_a : 16 * log(9)

Behavior Entropy for the top-level-entity half_adder_b  : 4 * log(9)
Structure Entropy for the top-level-entity half_adder_b : 6 * log(9)

Behavior Entropy for the top-level-entity half_adder_c  : 4 * log(9)
Structure Entropy for the top-level-entity half_adder_c : 18 * log(9)
```

Figure 6 charts the values for the behavior and structure entropy of all three realizations. As all designs describe a half-adder with two inputs and two outputs, the behavior entropy for all realizations is the same. Due to the difference in design, the structure entropy varies. The figures allow to give statements about the relative complexity of the three designs.

V. Conclusion and Future Work

We have introduced a tool, which is capable of analyzing hardware projects for their complexity. With this tool, it is possible to automatically calculate the complexity of VHDL implementations. The tool is written in Java and therefore platform independent. With this tool, we can now start to compare larger projects with each other. With these results, we can also evaluate the quality of the complexity measure method. As for now, our tool covers structural descriptions based on basic gates. We are working on a complete annotated grammar, which covers VHDL completely. Due to the enormous extent of the VHDL standard, we can only implement the syntax and the complexity calculation methods step by step. In this paper we were already able to show the beneficial influence of our tool for complexity calculations in digital hardware projects.

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, 1965.

[2] Intel Corporation, "Excerpts from A Conversation with Gordon Moore: Moore's Law," 2005. [Online]. Available: ftp://download.intel.com/museum/Moores_Law/Video-transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf

[3] M. Kanellos, "Moore's law to roll on for another decade," http://news.cnet.com/2100-1001-984051.html, 2003.

[4] G. E. Moore, "Progress in digital integrated electronics," vol. 21, 1975.

[5] R. Cavin, P. Lugli, and V. Zhirnov, "Science and Engineering Beyond Moore's Law," *Proceedings of the IEEE*, vol. 100, 2012.

[6] International Solid-State Circuit Conference, "Isscc 2013 trends report," 2013. [Online]. Available: http://isscc.org/doc/2013/2013_Trends.pdf

[7] T. DeMarco, *Controlling Software Projects: Management, Measurement, and Estimates*, 1986.

[8] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach, Revised.* Course Technology, 1998.

[9] B. Menhorn and F. Slomka, "Design entropy concept," in *ESWEEK 2011 Compilation Proceedings (CODES+ISSS '11)*, 2011.

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed., 2006.

[11] "IEEE standard VHDL language reference manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009.

[12] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, 1948.

[13] B. Menhorn and F. Slomka, "States and complexity," in *Coping with Complexity COPCOM 2011*, 2011.