Design Entropy Concept

A Measurement for Complexity

Benjamin Menhorn and Frank Slomka
Institute for Embedded Systems/Real-Time Systems
Ulm University, Germany
{benjamin.menhorn|frank.slomka}@uni-ulm.de

ABSTRACT

In general, this work will deal with measuring complexity. The focus question is towards addressing complexity in an adequate way. This work concentrates on digital circuits and digital hardware. For this field of computer science the complexity for circuits will be calculated.

Therefore, a new complexity measure will be introduced, called *design entropy*. It allows a mathematical calculation of complexity resulting in figures. These allow a direct evaluation and comparison between different systems and realizations. The application and important capabilities of this measurement will be demonstrated on different examples.

Categories and Subject Descriptors

B.6.m [**Hardware**]: LOGIC DESIGN—*Miscellaneous*

General Terms

Design, Measurement, Theory, Verification

Keywords

Complexity, Measurement, Entropy, Abstract, Model, States

Paper organization

This paper is organized as follows. Section 1 identifies a general need for a new and different measurement for complexity. The following section 2 describes the approach and the goals of the *design entropy concept*. Section 3 analyzes the origins for complexity and can deduce the formulas for the design entropy. Before section 5 presents the formulas of the concept section 4 will clarify some terminology. The final section 6 will apply the formulas on some different examples.

1. INTRODUCTION

Most engineering disciplines exert well defined methods and models to manage, control and evaluate projects. The determination of project size makes it possible to give statements such as needed resources for development, how development processes can be optimized and to compare different projects and implementations [3] [6]. For determining sizes of projects the main challenge is to get complexity under control.

Computer science especially digital circuit design is a very young science with only a few decades of experience. Additionally computer science is subjected to fast changes and developments in technology. Therefore empirical data from recent projects is hard to transfer to new projects and brings high inaccuracies [9].

Today, most methods for estimating project size use empirical data, by analyzing previous projects [8]. They try to find key figures with which project sizes can be estimated and compared. All those approaches are basically trying to get complexity under control. But most of these approaches only work for one certain technology, programming language or description language. This indicates a need for an abstract measurement, which can be used even with changes in technology and new developments.

For digital circuits, simply counting transistors was sufficient for an adequate estimation of project size in the beginning of micro electronics. Today, with rising complexities and sizes it is not enough to count transistors anymore [5]. In hardware design it is still possible to count transistors and maybe connections. But it wouldn't address complexity in an adequate way. With hardware description languages additional abstraction layers are introduced. This makes transistor counts very less significant as a complexity measurement. Hardware design today gets more and more equivalent to methods used in software engineering. If it would be possible to give funded complexity estimations, at the best represented by figures, project management models from other engineering disciplines could be used.

One way to achieve this goal, is to become independent from design methods and abstraction layers. Changes in technologies and new inventions would then still allow to use this model. And even more, a comparison between new and previous projects would still be possible. This calls for an abstract measurement.

2. DESIGN ENTROPY CONCEPT

It is important to have a model which can deal with different key aspects. For some projects key aspects are timing constraints, for others design costs and even others concentrate on optimization. But it would not be possible to have an applicable measurement for all concerns right away.

Therefore, this projects concentrates on digital circuits. Always in mind, that the developed formulas and applications methods should be also applicable within different

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

computer science disciplines. But even concentrating on digital circuits leaves a huge field for measuring complexity.

As an important first step, the measurement of the circuit's complexity was identified. At the beginning, this measurement could then be applied on small and midscale examples in order to review its success.

The goal is to have an abstract measurement, which could be applied on different abstraction levels. This would allow the concept to be applied on an early development stage. Projects could be planed using the complexity calculations. But for the first step, the analysis starts with given circuits or implementation codes.

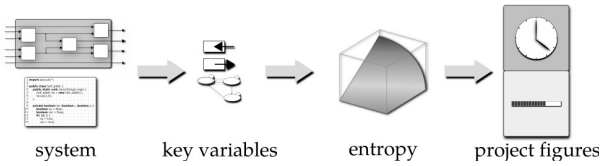

Figure 1: General approach

Figure 1 shows this concepts approach. For a given circuit, key variable values are determined. With these figures the complexity for this circuit can be calculated, which is called design entropy. The final concept will base statements about project specific figures such as duration, costs, progress and quality on the complexity values. But for now it should be sufficient to calculate complexity values for circuits.

The core of the model are states. 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. This indicates that entropy can be used to measure project size by using states. States are abstract variables which depend on the analyzed property of a project. They can be used for describing properties of hardware systems but also in other contexts. For example with software or embedded systems. States are abstract and do not directly depend on technologies, architectures and abstraction layers. These factors have an influence on the amount of possible states.

3. ORIGIN AND DEDUCTION

In order to understand why a model with abstract states as key variables could be able to address complexity in an adequate way, this section will discuss the origins of complexity. In particular the difference between regular and irregular structures. This will allow a conclusive explanation for using abstract states. The second half of this section introduces one part of Shannon's information theory. His formula provides the starting point for the design entropy formulas, which can be directly derived.

3.1 Regular and irregular structures

In context with integrated circuits, Moore's Law is often used to describe the development of complexity over time [10]. Originally complexity was measured by Gordon Moore in number of components per integrated function[11]. Later, transistors per chip were counted instead of components. The productivity measurement is related to transistors per day in the ITRS 2007 edition [2]. Figure 2 plots productivity and complexity over time. According to the ITRS 1999 edition the complexity has an annual growth rate of 58% while productivity has only a growth rate of 21% [1]. The diverge is called design gap. A common opinion found in literature bases the design gap on missing tools, abstraction layers and design possibilities (e.g. [7]: "system-level design and extensible processors can bridge the gap between silicon technology and actual SoC complexities"). This works advances the view that the design gap is originally caused by the difference between regular and irregular structures itself.

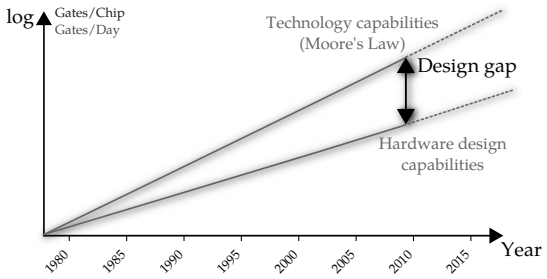

Figure 2: Design gap (Data from [4])

Memory, for example, is a regular structure for most parts. Figure 3 illustrates an eight-bit wide memory. Each square describes one memory cell and can hold one bit. With the development of only one (elementary) memory cell the basic component for a memory is designed. This cell can be copied in any quantity and put together to a grid. This grid and a separately developed control logic composes a whole memory. A simple control unit is only a (de-)multiplexer and displayed on the left hand side of figure 3. The ideas of developing one elementary cell and copying it in any quantity comes together with the idea of reusing components. It reduces the design complexity drastically.

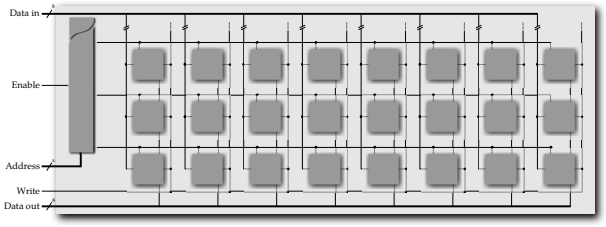

Figure 3: Memory

In contrast to memory, processors are at most irregular structures. Of course, processors are not completely irregular structures. For instance, adders can be reused and processor internal cache also consists of single memory cells. Processor control logics are a highly irregular structure which is harder to design. There is no elementary cell that can be copied in any quantity. By adding regular structures, for example by increasing the processor's memory (cache), the throughput can be increased. By reducing the complexity of irregular structures at the same time, e.g. by using RISC (**R**educed **I**nstruction **S**et **C**omputer) architectures, the ratio between regular and irregular structures can be enhanced in favor of less complex designs.

Smaller elements allow a higher gate (transistor) density. The same chip area then comprises more elements. Having more elements on the same chip area leads to more connection possibilities for the same area. Filling an area with regular structures in less complex. Filling the same area with irregular structures requires more time and effort.

Moor's Law describes the technical possible or in other words the highest gates/transistors density. The highest

gate density can be found in memories, an highly regular structure. But chip design comprises more than adding regular structures together. Designing irregular structures leads to a slower productivity growth because of the higher complexity.

Defining complexity over connections and states allows to consider this divergence. Building memory cells reduces the connection possibilities of the including gates. Only the cells have to be connected. There are less possible states the system (in this case the chip design) can presume. When building a processor control unit there are almost no elementary cells. Therefore, there are many possibilities to connect gates. This increases complexity by having a larger amount of possible states.

Additional abstraction layers and design tools shorten the gap between technology capabilities and design capabilities. Those approaches reduce the decision possibilities for designers and engineers. Thereby the amount of possible states of a system is reduced. But the cause of the design gap is due to the difference between regular and irregular structures itself.

3.2 Shannon's information entropy

The formulas for the design entropy concept can be derived from Shannon's information entropy. Signals between components can be seen as a transmission of information. Connections are the channel and information is symbols transfered from a pool of available symbols. Connections allow to interchange information. In digital hardware connections are normally realized by wires. In software information interchange can happen through assignments, calls or statements, for instance. But it is still transmission of information between components. For instance an assignment between two variables: `a := b`: The information *(=value)* from component *(=variable)* `b` is transmitted *(=assigned)* to component *(=variable)* `a`.

In order to give mathematical statements about transmitted information, Claude Elwood Shannon developed a model which became famous as Shannon's information theory [12]. The form of his theorem (see (1)) is recognized as that of entropy as defined in certain formulations of statistical mechanics (e.g. [13]), where p_α is the probability of a system being in cell α of its phase space. H is from Boltzmann's famous H theorem and the constant K merely amounts to a choice of a unit of measure [12]. According to C. E. Shannon, J. W. Tukey suggested to call the information content $I = 1$ bit for devices with two stable positions. This work doesn't assign a unit to the calculated entropy.

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

For calculating complexity only the maximum entropy is relevant. Entropy is used as a measurement for complexity. Intuitively, complexity is larger in uncertain situations. Therefore it is only of interest, what the maximal complexity for a component is. The entropy becomes maximal in case all possibilities are uniformly distributed. Figure 4 plots the entropy as a function of p in case of two possibilities with probabilities p and $q = 1 - p$. The entropy is given by: $H = -(p \log p + q \log q)$. As expected the entropy becomes maximal when $p = q = 0.5$ holds. In this case the possibilities are uniformly distributed.

In (1) p_α is the probability that the elementary event α occurs. H has its maximum, if every elementary event can occur with the same probability. Then all p_α are uniformly distributed and $p_\alpha = 1/N$ holds. Equation (1) can be rewritten to (2).

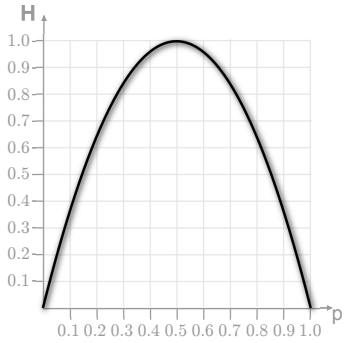

Figure 4: Entropy for two possibilities

$$H_{\max} = -\sum_{i=1}^{N} \frac{1}{N} \log \frac{1}{N} = \log N \tag{2}$$

In (2), N is the amount of possible symbols which can be transmitted. For the transmission of information between components N is the number of possible states. The channel between components are connections. Figure 5 shows two components. These two components have three connections. Each connection can transmit a certain number of different states (z_1, z_2 and z_3).

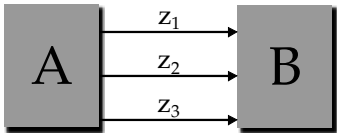

Figure 5: Channel between two components

The first channel has z_1 possible input symbols (states). Therefore the maximum of $H(z_1)$ is $H(z_1) = \log N = \log z_1$. For determining the entropy of all connections between component A and B, all three channels have to be considered. The joint entropy is then the sum of the single entropies $H(z_1)$, $H(z_2)$ and $H(z_3)$.

$$\begin{aligned} H(A \to B) &= H(z_1) + H(z_2) + H(z_3) \\ &= \log(z_1) + \log(z_2) + \log(z_3) \\ &= \log(z_1 \cdot z_2 \cdot z_3) \end{aligned}$$

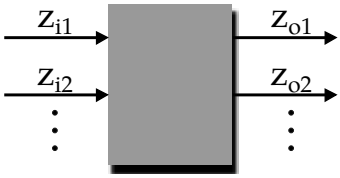

Figure 6: Connections of a component

In order to calculate the entropy for one component, in- and outputs have to be considered. Figure 6 illustrates a component. The entropy for the whole component is then

given by the joint entropy of all in- and outputs.

$$\begin{aligned} H(\text{component}) =& H(\text{inputs}) + H(\text{outputs}) \\ =& \log(z_i 1) + \log(z_i 2) + \ldots \\ & + \log(z_o 1) + \log(z_o 2) + \ldots \\ =& \log(z_i 1 \cdot z_i 2 \cdot \ldots \\ & \cdot z_o 1 \cdot z_o 1 \cdot \ldots) \\ =& \log\left(\prod_{i \in \text{inputs}} z_i \cdot \prod_{j \in \text{outputs}} z_j \right) \end{aligned}$$

With the annotation for inputs from definition 5, for outputs from definition 6 and for states from definition 7 the general formula for the behavior entropy from definition 8 follows in (3).

$$H_B(c) = \log\left(\prod_{i=1}^{n(c)} z(n_i(c)) \cdot \prod_{j=1}^{m(c)} z(m_j(c)) \right) \qquad (3)$$

4. TERMINOLOGICAL CLARIFICATIONS

Since this work is addressed to an interdisciplinary and heterogeneous audience, it seems advisable to pay attention to some terminological clarifications. This ensures that crucial terms which appear have the same sense through different communities. These definitions are taken as a basis for the following section, where the design entropy concept will be defined.

4.1 Component, behavior and structure

The *design entropy* formulas are always applied on components. Therefore definition 1 provides information about the term component. Depending on the context, a component can also be called entity, block, class, function or procedure. The core of the definition about components is that projects can be partitioned. Those parts are closed projects for themselves and called components. Definitions 2 to 4 define the terms behavior, structure and verification complexity.

DEFINITION 1 (COMPONENT). *A component is either the highest part of a project (e.g. top-level entity or main class) or a closed part of a project with a defined (non infinite) number of in- and outputs.*

DEFINITION 2 (BEHAVIOR COMPLEXITY). *The behavior complexity (also called behavior entropy) gives a statement about the (usage) complexity of a component.*

DEFINITION 3 (STRUCTURE COMPLEXITY). *The structure complexity (also called structure entropy) gives a statement about the implementation/realization complexity of a component.*

The behavior complexity does not allow for the actual implementation complexity of a component. It only provides complexity information about the usage of a component. It can be compared to an outer or black box view on a component. In contrast to the behavior complexity the structure complexity provides information how complex the implementation/realization of a component is. This is similar to an inner or white box view on a component.

But the terms inner and outer view are not adequate. For the entropy calculations states of components are considered. Therefore the behavior complexity does on a certain scale allow also a look inside. Consequently it is not completely an outer look. Also an inner view does not completely dissolve all inner operations. Components can have closed sub components inside which are only considered by their behavior entropy. Using the terms black and white box would lead to confusion with verification in software engineering. Therefore, based on hardware design descriptions, the terms behavior and structure are used. These terms address the difference and intend in an adequate way.

In general, the behavior complexity is used for instances of implemented components. The structure complexity is used to compare different realizations or different projects. In order to determine the structure complexity of a component, all including parts have to be considered. These parts are mainly sub components with an own behavior complexity. If these sub components have also been implemented, their structural complexity needs to be considered, too. In summary: everything implemented within a project and all instances of components, disregarding if implemented or reused, are considered for the structure complexity.

This section provides its last definition for the verification complexity. As the term verification already indicates, this measurement provides information about the complexity to verify a component.

DEFINITION 4 (VERIFICATION COMPLEXITY). *The verification complexity (also called verification entropy) gives a statement on the complexity to verify all possible states of a component and its parts.*

Before giving the formulas for the design entropy in section 5, the following short part completes the preliminary definitions.

4.2 Inputs, outputs and states

A component can have inputs, annotated with n. $n(c)$ is the amount of inputs of component c. A subscript indicates the single inputs. For instance $n_1(c)$ is the first input from component c. Considering $n(c)$ inputs, $n_{n(c)}(c)$ is the name for the last input. Outputs are defined analog with the letter m instead of n.

This leads to three definitions: definition 5 for the inputs, definition 6 for the outputs and definition 7 for the states of the in- and outputs of a component.

DEFINITION 5 (INPUTS). *For a given component c, $n(c) \in \mathbb{N}_0$ is the amount of inputs (sources) of component c. The inputs are enumerated and entitled $n_i(c), i = 1, \ldots, n(c)$ for a component c with $n(c)$ inputs.*

DEFINITION 6 (OUTPUTS). *For a given component c, $m(c) \in \mathbb{N}_0$ is the amount of outputs (drains) of component c. The outputs are enumerated and entitled $m_j(c), j = 1, \ldots, m(c)$ for a component c with $m(c)$ outputs.*

Every in- and output needs to have a defined, non zero and not infinite number of states. Each state has to be differentiable from each other state of the same in- or output. Figure 7 illustrates how states for each in- and output are annotated[1]. In this example the first input $n_1(c)$ and the first output $m_1(c)$ have each two possible states: 0 or 1 (also called *true* or *false*). The second input $n_2(c)$ and the second output $m_2(c)$ have four possible states: in this example 00,

[1]Allocations of in- and outputs with the component have been skipped in figure 7 in interest of readability. The complete annotation should be $n_1(c) \ldots n_{n(c)}(c)$ instead of only $n_1 \ldots n_n$. Same for outputs of c with m.

01, 10 or 11. It is not important how single states look like, only the amount of possible states is important. This leads to definition 7 about the amount of possible states.

Definition 7 (States). *For a given component c with $n(c)$ inputs and $m(c)$ outputs, the amount of possible states for the inputs is given by $z(n_i(c)) \in \mathbb{N}$, $i = 1, \ldots, n(c)$ and for the outputs by $z(m_j(c)) \in \mathbb{N}$, $j = 1, \ldots, m(c)$. The amount of states has to be not infinite: $z(n_i(c)) < \infty$ and $z(m_j(c)) < \infty$.*

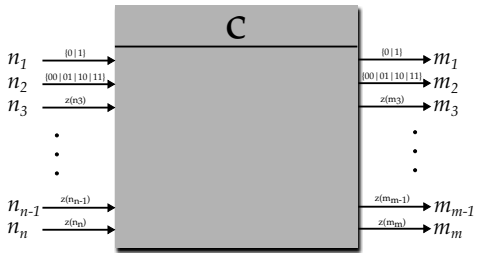

Figure 7: States of the in- and outputs from component c

5. DESIGN ENTROPY DEFINITIONS

The following definitions provide the framework for the *design entropy concept*. Using the defined (key-)variables from the previous section 4, all necessary information for calculating the behavior, structur and verification entropy is given. The most abstract and general statement about a component is provided by the behavior entropy in definition 8. It is the same formula as deduced from Shannon's information entropy in equation (3).

Definition 8 (General 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:*

$$H_B(c) = log\left(\prod_{i=1}^{n(c)} z(n_i(c)) \cdot \prod_{j=1}^{m(c)} z(m_j(c))\right) \tag{4}$$

In case a component has a constant number of states for all in- and outputs ($z(c) = z(n_i(c)) = z(m_j(c)) \; \forall i \in \{1, \ldots, n(c)\}$ &$\forall j \in \{1, \ldots, m(c)\}$), equation (4) can be rewritten:

$$\begin{aligned} H_B(c) &= \log\left(\prod_{i=1}^{n(c)} \underbrace{z(n_i(c))}_{=z(c)} \cdot \prod_{j=1}^{m(c)} \underbrace{z(m_j(c))}_{=z(c)}\right) \\ &= \log\left(\prod_{i=1}^{n(c)} z(c) \cdot \prod_{j=1}^{m(c)} z(c)\right) \\ &= \log\left(\underbrace{\prod_{i=1}^{n(c)} z(c)}_{=z(c)^{n(c)}} \cdot \underbrace{\prod_{j=1}^{m(c)} z(c)}_{=z(c)^{m(c)}}\right) \\ &= \log\left(z(c)^{n(c)+m(c)}\right) \\ &= (n(c) + m(c)) \cdot \log(z(c)) \end{aligned}$$

Rewriting the equation leads to definition 9. This equation for the behavior entropy can be applied in case all connections (in- and outputs) have an equal amount of states. Especially in hardware and integrated circuit designs, this formula becomes dominating. Most connections between components will have only two states: *high* or *low*. Therefore in- and outputs will also have only two possible states.

Definition 9 (Behavior Entropy). *Let c be a component with $n(c)$ inputs (component sources) and $m(c)$ outputs (component drains). Each in- and output has $z(c)$ possible states. Then the behavior entropy $H_B(c) \in \mathbb{R}_0^+$ of a component c is defined as:*

$$H_B(c) = (n(c) + m(c)) \cdot log(z(c)) \tag{5}$$

Equation (5) illustrates that the behavior entropy is the sum of all in- and output states. This expresses the number of states a component. Because the behavior entropy has an outer look on components and describes the complexity for using components. The structure entropy allows for the implementation of components and has therefore an inner look. This statement is also valid for the general calculation of the behavior entropy from equation (4) but it seems more clear with the reduced equation (5). The amount of states a component can have is the source for a higher or lower complexity. Influencing this amount of possible states allows controlling complexity. But in order to calculate the complexity of a whole project with sub components it is necessary to have the ability to treat those sub components adequately. Which means that it is necessary to distinguish between implemented/realized components and instances of components. With this distinction the structure entropy from the following definition 10 provides the ability to calculate the structure complexity of a component. This expresses the complexity for realizing this component.

Definition 10 (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:*

$$H_S(c) = \sum_{i \in c_b} H_B(i) + \sum_{j \in c_s} H_S(j) \tag{6}$$

Formula (6) realizes the recursive approach. The analysis of a project normally starts at the highest level. The structure entropy sums up the structure entropy of all direct sub

components. Those sub components can have sub components with structure entropies, too. In order to determine the structure entropy of the whole project it is necessary go through the project and calculate all structure entropies.

This is similar to tree traversals. The project has to be parsed from the "root node" to all "leaves". All components with only a behavior entropy build leaves and all components with structure entropies build the notes of the tree.

This recursive approach expressed in (6) is the modular part of the design entropy concept. It allows to partition the project. Reusability is supported by the difference between behavior and structure entropy.

In order to give a measurement for the verification complexity, the following formula is proposed. Theoretical results have shown that complexity in model-checking is exponential proportional to the amount of properties to verified. There are many publications, models and proposals about the verification of a system. Therefore, the verification entropy proposed here is an idea how to calculate the complexity of a verification. It bases on the idea of the design entropy concept but does not belong to the core concept. It is assumed that components have only to be checked once for their working correctness. More instances of the same component do not have to be checked again. It is therefore sufficient to verify that all distinct sub components are working correctly. Finally only the connections have to be checked, which is done by verifying the component which contains the sub components.

Definition 11 (Verification Entropy). *Let c be a component with different sub components c_d. Then the verification entropy $H_V(c) \in \mathbb{R}_0^+$ for component c is given by the sum of all behavior entropies of all different (distinct) instances c_d and the behavior entropy of the components c:*

$$H_V(c) = \sum_{i \in c_d} H_B(i) + H_B(c) \tag{7}$$

Figure 8 illustrates how definition 11 applies to an abstract project. Components which have to be verified are gray colored in the figure. The white colored components don't need to be verified, because another component, which is an instance of the same component, was already verified. The

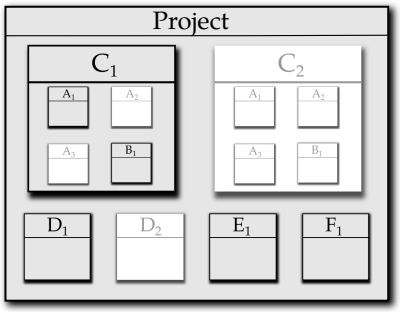

Figure 8: Verifying components of an abstract project

projects from figure 8 includes two components C_1 and C_2 which have sub components A_1, A_2, A_3 and B_1. For verifying C_1 it is necessary to verify A_1, B_1 and the component C_1 itself. A_1, A_2 and A_3 are instances of the same component. Therefore it is sufficient to verify only one instance. This assures that the component is working correctly and all instances, too. It can happen that connections between components are wrong. A correct connection can be verified by verifying the component C_1 itself. Because C_1 has already been verified C_2 doesn't need to be verified again because it is a copy of C_1. In order to verify the whole project the following parts have to be verified: A_1, B_1, C_1, D_1, E_1, F_1 and the project itself.

6. CAPABILITY DEMONSTRATION

For a better understanding of the design entropy concept this sections gives an example how to analyze a ripple-carry-adder and demonstrates the capabilities of the design entropy concept. This example was chosen ensuring every reader is able to follow the approach and calculations. First, different realizations of half-adders are analyzed and compared. Half-adders can then be used to build full-adders. By having four instances of a full-adder, a ripple-carry-adder can be designed. For each design step, complexity statements will be given. This component based approach shows the modularity, capability and the ability to reuse components.

This section will directly show that the design entropy concept is able to calculate complexity and express it in figures. Most of the results are those expected, because most readers will have a sense for complexity. But instead of making complexity statements based on sense or experience, the design entropy concept is able to express complexity in figures. This allows a direct comparison of different realizations and projects. For more complex designs a statement about complexity based on a designer's sense or experience becomes more and more impossible. Therefore an example was chosen which can be followed by all readers.

6.1 Half-adder

There are several ways to realize a half-adder. Within this example, three different realizations are analyzed and compared. For each implementation, the structure, the behavior and the verification entropy are calculated. This allows a comparison of all three implementations.

It can be assumed that logic gates are already implemented/provided. According to definition 10, for the structure entropy only instances of those gates have to be considered in order to calculates the structure entropy of the half-adders. All in- and outputs from the logic gates and the half-adders have the same amount of states. It is sufficient to use the behavior entropy calculation for components with an equal amount of states from equation (5). Except for inverters, all basic gates have two inputs and one output. Therefore the behavior entropy for all those gates is the same and can be calculated according to equation (5):

$$\begin{aligned} k &\in \{\text{AND}|\text{NAND}|\text{OR}|\text{NOR}|\text{XOR}|\text{XNOR}\} \\ H_B(\text{k}) &= (n(\text{k}) + m(\text{k})) \cdot \log(z(\text{k})) \\ &= (2+1) \cdot \log(2) \\ &= 3 \cdot \log(2) \end{aligned}$$

Inverters have only one input and one output. Compared to gates with two inputs, inverters are less complex and have a lower behavior entropy.

$$\begin{aligned} H_B(\text{Inverter}) &= (n(\text{Inverter}) + m(\text{Inverter})) \cdot \log(z(\text{Inverter})) \\ &= (1+1) \cdot \log(2) \\ &= 2 \cdot \log(2) \end{aligned}$$

6.1.1 Realization A

One way to realize a half-adder is with two inverters, three AND gates and one OR gate. Figure 9 illustrates this realization. This allows the calculation of the behavior entropy according to formula (5), the structure entropy according to formula (6) and the verification entropy according to formula (7) of realization A:

$$\begin{aligned} H_B(\text{half-adder}_\text{A}) =& (n(\text{half-adder}_\text{A}) + m(\text{half-adder}_\text{A})) \\ & \cdot \log(z(\text{half-adder}_\text{A})) \\ =& (2+2) \cdot \log(2) \\ =& 4 \cdot \log(2) \end{aligned}$$

$$\begin{aligned} H_S(\text{half-adder}_\text{A}) =& 2 \cdot H_B(\text{Inverter}) + 3 \cdot H_B(\text{AND}) \\ & + H_B(\text{OR}) \\ =& 2 \cdot 2 \cdot \log(2) + 3 \cdot 3 \cdot \log(2) + 3 \cdot \log(2) \\ =& 16 \cdot \log(2) \end{aligned}$$

$$\begin{aligned} H_V(\text{half-adder}_\text{A}) =& H_B(\text{Inverter}) + H_B(\text{OR}) \\ & + H_B(\text{AND}) + H_B(\text{half-adder}_\text{A}) \\ =& 2 \cdot \log(2) + 3 \cdot \log(2) + 3 \cdot \log(2) \\ & + 4 \cdot \log(2) \\ =& 12 \cdot \log(2) \end{aligned}$$

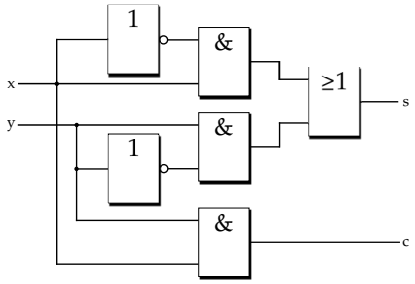

Figure 9: Realization A of a half-adder with AND, OR and Inverter gates

6.1.2 Realization B

Some occasions make it necessary to have realizations with only one type of gate, for instance when working with programmable NAND or NOR logic arrays. A half-adder with only NAND gates could be realized as shown in figure 10. This leads to the following entropies:

$$\begin{aligned} H_B(\text{half-adder}_\text{B}) =& (n(\text{half-adder}_\text{B}) + m(\text{half-adder}_\text{B})) \\ & \cdot \log(z(\text{half-adder}_\text{B})) \\ =& (2+2) \cdot \log(2) \\ =& 4 \cdot \log(2) \end{aligned}$$

$$\begin{aligned} H_S(\text{half-adder}_\text{B}) &= 6 \cdot H_B(\text{NAND}) \\ &= 6 \cdot 3 \cdot \log(2) \\ &= 18 \cdot \log(2) \end{aligned}$$

$$\begin{aligned} H_V(\text{half-adder}_\text{B}) &= H_B(\text{NAND}) + H_B(\text{half-adder}_\text{B}) \\ &= 3 \cdot \log(2) + 4 \cdot \log(2) \\ &= 7 \cdot \log(2) \end{aligned}$$

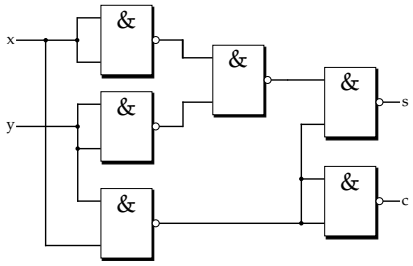

Figure 10: Realization B of a half-adder with six NAND gates

6.1.3 Realization C

The simplest way with the least amount of gates is a realization with one AND and one XOR gate. Both gates have two inputs and one output. Figure 11 illustrates this realization. Now calculating the behavior, structure and verification entropy of realization C:

$$\begin{aligned} H_B(\text{half-adder}_\text{C}) =& (n(\text{half-adder}_\text{C}) + m(\text{half-adder}_\text{C})) \\ & \cdot \log(z(\text{half-adder}_\text{C})) \\ =& (2+2) \cdot \log(2) \\ =& 4 \cdot \log(2) \end{aligned}$$

$$\begin{aligned} H_S(\text{half-adder}_\text{C}) &= H_B(\text{AND}) + H_B(\text{XOR}) \\ &= 3 \cdot \log(2) + 3 \cdot \log(2) \\ &= 6 \cdot \log(2) \end{aligned}$$

$$\begin{aligned} H_V(\text{half-adder}_\text{C}) =& H_B(\text{AND}) + H_B(\text{XOR}) \\ & + H_B(\text{half-adder}_\text{C}) \\ =& 3 \cdot \log(2) + 3 \cdot \log(2) + 4 \cdot \log(2) \\ =& 10 \cdot \log(2) \end{aligned}$$

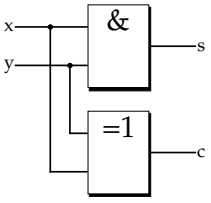

Figure 11: Realization C of a half-adder with one AND and one XOR gate

6.1.4 Comparison of realization A,B,C

Figure 12 charts the values for the behavior, structure and verification entropy of all three realizations. It is not surprising that the behavior entropy is the same for all. All three realizations have the same amount of in- and outputs

as well as the same amount of states for each connection. Therefore the behavior entropy has to be the same.

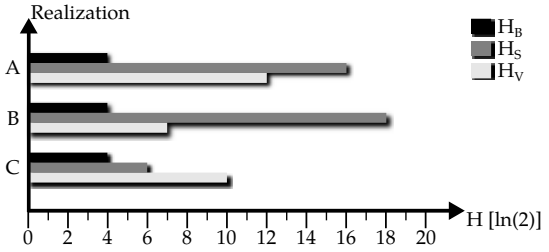

Figure 12: Entropy of the three half-adder realizations

The structure and verification entropy is directly influenced by the amount and behavior entropy of the used gates. The easiest way from the given choice to implement a half-adder is realization C with only one XOR and one OR gate. This is plausible and the design entropy concept was able to express it through a mathematical calculation resulting in figures. While realization B only uses NAND gates, realization C needs two different gates and realization A three different gates. Therefore A has the highest verification entropy. Both, realization A and B, have the same amount of gates. But inverters, which are used in realization A, have a lower behavior entropy than the other gates with two inputs. Therefore A has a lower structure entropy then B.

6.2 Full-adder

There are several goals for the management of a project. An often quoted one is saving development time. A short product development cycle allows an early market entry. A project can be developed faster when the complexity is lower: there are less decision possibilities and less errors to make. In order to accomplish this goal realization C from figure 11 with one AND and one XOR gate is used to build a full-adder. This realization had the lowest structure entropy.

A full-adder consists of two half-adders and one OR gate, as illustrated in figure 13. This allows the calculation of the behavior entropy and the structure entropy of the half adder.

$$\begin{aligned} H_B(\text{full-adder}) =& (n(\text{full-adder}) + m(\text{full-adder})) \\ & \cdot \log(z(\text{full-adder})) \\ =& (3+2) \cdot \log(2) \\ =& 5 \cdot \log(2) \end{aligned}$$

$$\begin{aligned} H_S(\text{full-adder}) =& 2 \cdot H_B(\text{half-adder}) + H_B(\text{OR}) \\ & + H_S(\text{half-adder}) \\ =& 2 \cdot 4 \cdot \log(2) + 3 \cdot \log(2) + 6 \cdot \log(2) \\ =& (8+3+6) \cdot \log(2) \\ =& 17 \cdot \log(2) \end{aligned}$$

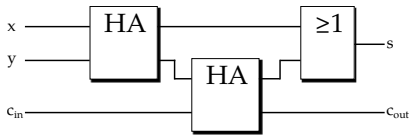

Figure 13: Full-adder

6.3 Ripple-carry-adder

Using the full-adder from figure 13 and having four instances, a ripple-carry-adder can be implemented as illustrated in figure 14. The behavior and structure entropy is then given by:

$$\begin{aligned} H_B(\text{rca}) &= (n(\text{rca}) + m(\text{rca})) \cdot \log(z(\text{rca})) \\ &= (9+5) \cdot \log(2) \\ &= 14 \cdot \log(2) \end{aligned}$$

$$\begin{aligned} H_S(\text{rca}) &= 4 \cdot H_B(\text{full-adder}) + H_S(\text{full-adder}) \\ &= 4 \cdot 5 \cdot \log(2) + 17 \cdot \log(2) \\ &= (20+17) \cdot \log(2) \\ &= 37 \cdot \log(2) \end{aligned}$$

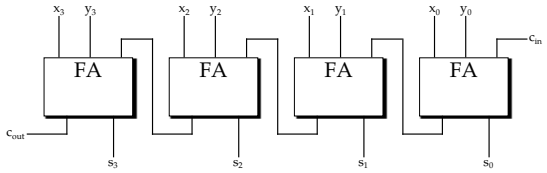

Figure 14: Ripple-carry-adder realized with full-adders

This ripple carry-adder could now be used to build a whole CPU, for instance. But for now, it was sufficient to analyze the components of a ripple-carry-adder starting with logic gates. Thereby this example was able to demonstrate the capabilities of the modular approach of this concept. The rest of this section will calculate with different amounts of states and show the effect of changing the design and partitioning.

6.4 Ripple-carry-adder with different amounts of states

In the previous calculations of this example, it was always assumed that every in- or output has only two possible states. This short section will deal with connection that have different amounts of states. This situation will become standard when analyzing software. In order to demonstrate (4) for the general behavior entropy, the ripple-carry-adder from the previous section is used (figure 14). Digital circuits allow to write the amount of possible states at an in- or output to a number to the power of two. This allows a followable comparison between an approach with equal amounts of states and different amounts of states at all in- and outputs.

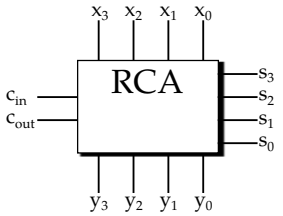

Figure 15: Ripple-carry-adder with one bit connections

For now, the ripple-carry-adder from the previous section had in- and outputs with only two possible states. This is illustrated in figure 15. In hardware design (languages)

Figure 16: Ripple-carry-adder with one and four bit connections

there is also the possibility to have logic vectors with more than two states. For a ripple-carry-adder there could be two vectors $\mathbf{x}$ and $\mathbf{y}$ with each four bits for the input, an one bit input c_{in} for carry in, a four bit output vector $\mathbf{z}$ for the sum and an one bit output c_{out} for carry out. Figure 16 illustrates a ripple-carry-adder with vectors. Vectors $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ have each 16 different states. In order to calculate the behavior entropy now equation (4) is used. The reduced equation (5) for the behavior entropy can't be applied anymore.

$$\begin{aligned} H_B(\text{rca}) &= \log\left(\prod_{i=1}^{n(\text{rca})} z_i(\text{rca}) \cdot \prod_{j=1}^{m(\text{rca})} z_j(\text{rca})\right) \\ &= \log\left(16 \cdot 16 \cdot 2 \cdot 16 \cdot 2\right) \\ &= \log\left(16384\right) \\ &= \log\left(2^{14}\right) \\ &= 14 \cdot \log\left(2\right) \end{aligned}$$

Calculating the behavior entropy of the same component brought the same results. No matter if connections with only two possible states are used or vectors with different amounts of states. In most hardware design analyses the number of states can be reduced to a number to the power of two. In software design analyzes it is common to have distinct amounts of states for all in- and outputs.

6.5 NAND ripple-carry-adder

It was already mentioned before that design decisions sometimes make it necessary to use only one certain type of gate. For example when there is only a chip with programmable NAND gates available. For demonstrating the influence of design decisions, a ripple-carry-adder with only NAND gates will be analyzed. Therefore the half-adder and the full-adder design have to be adjusted. A half-adder with only NAND gates has already been analyzed in section 6.1.2. The OR gate within the full adder has to be replaced by three NAND gates according to figure 17. This leads to the following entropy for the full-adder and finally for the ripple-carry-adder.

$$\begin{aligned} H_S(\text{full-adder}) =& 2 \cdot H_B(\text{half-adder}) + 3 \cdot H_B(\text{NAND}) \\ &+ H_S(\text{half-adder}) \\ =& 2 \cdot 4 \cdot \log(2) + 3 \cdot 3 \cdot \log(2) + 18 \cdot \log(2) \\ =& (8 + 9 + 18) \cdot \log(2) \\ =& 35 \cdot \log(2) \end{aligned}$$

$$\begin{aligned} H_S(\text{ripple-carry-adder}) &= 4 \cdot H_B(\text{full-adder}) + H_S(\text{full-adder}) \\ &= 4 \cdot 5 \cdot \log(2) + 35 \cdot \log(2) \\ &= (20 + 35) \cdot \log(2) \\ &= 55 \cdot \log(2) \end{aligned}$$

A ripple-carry-adder with only NAND gates has a higher entropy, here $55 \cdot \log(2)$. The first realization of a ripple-carry-adder had a structure entropy of $37 \cdot \log(2)$ (see section 6.3). This increase is directly related to the higher number of

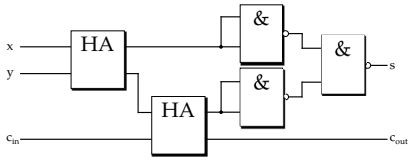

Figure 17: Full-adder with NAND gates

used gates. Instead of total 20 gates, were 60 NAND gates necessary. This increase of gates allows more connection possibilities and therefore more possible states. It results in a higher structure entropy. But the increase is not linear because the project was partitioned. It was more complex to built a half-adder with NAND gates. Therefore the half-adder has a higher structure entropy. For the full-adder, instances were used. Therefore the behavior entropy was used, which is the same for all half-adder realizations from section 6.1. The following section will show the impact on a project's complexity when turning down components.

6.6 Full-adder without components

Almost all hardware and software designer agree that partitioning a project in "handable" and "overlookable" parts is a key for a successful project. This section demonstrates the increase in complexity for a full-adder when no components are used.

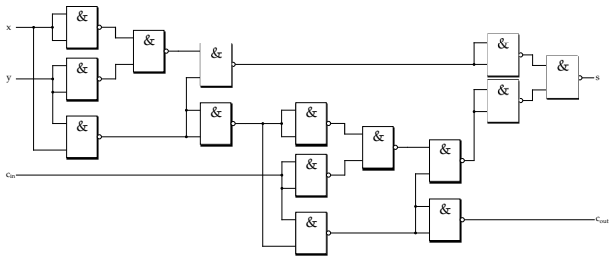

Figure 18: NAND Full-adder without components

Figure 18 illustrates a full adder, based on figure 17. Instead of using half-adders as components this realization does not have any sub components and includes all gates on the highest level.

$$\begin{aligned} H_S(\text{full-adder}) &= 15 \cdot H_B(\text{NAND}) \\ &= 15 \cdot 3 \cdot \log(2) \\ &= 45 \cdot \log(2) \end{aligned}$$

With half-adder components the structure entropy was $35 \cdot \log(2)$. Working without components increases the structure entropy by $10 \cdot \log(2)$. It lend itself to work with components, which can be reused, as long as they are not to small and would increase the complexity.

6.7 8-bit processor

This last application will demonstrate that the concept can also be applied to larger projects, in this case an eight bit processor. Such a processor could look like the one in figure 19. The processor has a separate instruction and data memory. The instruction memory has the width of 16 bit in order allow instructions to be wide enough for a complete data memory address or to jump to an address in the instruction memory.

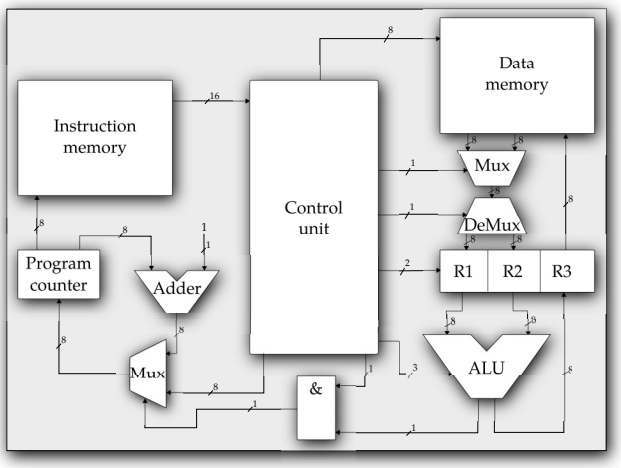

Figure 19: 8-bit wide processor

The program counter is a simple 8 bit wide register which can be loaded with the next addresses from the adder (+1) or a direct jump address from the control unit. The registers before and after the ALU (**A**rithmetic **L**ogic **U**nit) are controlled by the control unit and can be loaded from the data memory or with data from the control unit. This source selection is done by the multiplexer (Mux). The target selection of register 1 or 2 is done by a de-multiplexer (DeMux). The ALU result is written back to register 3 and from there to the data memory. The result can also be used to decide a conditional jump.

In order to calculate the entropy of the whole processor, the entropy for the single components has to be summed up. Applying (5) on every single component in conjunction with (6) allows to calculate the entropy of the whole processor.

$$H_S(\text{processor}) = \sum_{i \in \text{Components}} H_B(i) = (32 \cdot 8 + 21) \cdot \log(2) = 277 \cdot \log(2) \quad (8)$$

7. CONCLUSION

This paper's objectives were to introduce the *design entropy concept* and demonstrate its capabilities. At the beginning of this work it has been shown that the need for a new complexity measure is motivated by the rapid progress in computer science. Measurements, which are based on empirical data, are hard to apply to current projects and bring high inaccuracies. Furthermore, the new measure is placed on the claim that it allows comparisons between projects, which were implemented in different technologies and on different levels of abstraction. Considering that quite different aspects of a project should be considered, implies the need for an abstract model which uses an abstract key variable. These general abilities were achieved by using *states* as one key variable.

The formulas for calculating complexity could be derived by resorting to Shannon's information theory. In basis of his work, the calculated complexity is called *entropy* and the whole project design entropy concept. The concept comprises formulas for calculating a behavior, a structure and a verification entropy.

The presented formalism and formulas were then applied to the field of digital circuits and digital hardware. The applications presented here provided comprehensible examples, which have shown the following possibilities of the concept:

- Complexity can be calculated and expressed by figures.
- Implementations can be compared directly.
- The concept allows a project to be split into components.
- The concept takes reusability of components into account.
- Changes in complexity can be seen directly when a design is changed.

Future work will expand the concept further. It will not only be applied to digital circuits but also to software projects as well as to embedded systems. Furthermore, tools will allow an automated calculation of complexity. This will allow larger case studies and comparisons of (larger) real world projects.

8. REFERENCES

[1] Association, s. International technology roadmap for semiconductors. http://public.itrs.net/files/1999_SIA_Roadmap/Home.htm, 1999.

[2] Association, s. International technology roadmap for semiconductors. http://www.itrs.net/Links/2007ITRS/Home2007.htm, 2007.

[3] DeMarco, T. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

[4] Ecker, W., Müller, W., and Dömer, R. *Hardware-dependent Software - Principles and Practice*. Springer, 2006.

[5] Ermolayev, V., and Matzke, W.-E. Towards industrial strength business performance management. In *HoloMAS '07: Proceedings of the 3rd international conference on Industrial Applications of Holonic and Multi-Agent Systems* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 387–400.

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

[7] Henkel, J. Closing the soc design gap. *Computer 36*, 9 (2003), 119–121.

[8] Hinrichs, N., Leppelt, P., and Barke, E. Building up a performance measurement system to determine productivity metrics of semiconductor design projects. In *IEEE International Engineering Management Conference (IEMC), Austin TexasErmolayev2007* (2007), IEEE, Ed., IEEE, pp. CD–ROM Proceedings.

[9] Leppelt, P., Hassine, A., and Barke, E. An approach to make semiconductor design projects comparable. In *7th Asia Pacific Industrial Engineering and Management Systems Conference (APIEMS 2006)* (2006), Asian Institute of Technology, pp. CD–ROM.

[10] Mollick, E. Establishing moore's law. *IEEE Annals of the History of Computing 28*, 3 (2006), 62–75.

[11] Moore, G. E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (April 1965).

[12] Shannon, C. E. A mathematical theory of communication. *Bell System Technical Journal 27* (1948), 379–423, 623–656.

[13] Tolman, R. C. *The principles of statistical mechanics*. Oxford Univ. Pr., London, 1938.