QUANTITATIVE ANALYSIS OF SOFTWARE CODE BY STATES

Benjamin Menhorn and Frank Slomka
Institute for Embedded Systems/Real-Time Systems
Ulm University
Albert-Einstein-Allee 11, Ulm, Germany
{benjamin.menhorn|frank.slomka}@uni-ulm.de

ABSTRACT
In software engineering the determination of software size is a central issue. There have been numerous publications about software size estimation methods. Although some of these methods provide very good results in specific areas, there is no method that is both abstract enough to be used on different areas and at the same time addresses complexity in an adequate way.

This work presents a new approach for analyzing software code quantity by using states. Thereby states provide an abstract variable to measure complexity and a mathematical model calculates complexity. The mathematical model is based on Shannon's information entropy. This method has successfully been applied in the field of hardware engineering. We will now show, how this abstract method can also be used to calculate the size of software. With states it is possible to calculate complexity of software which we call entropy. A system's disorder is a property of a system's state. Therefore complexity is a measurement for a system's disorder. By changing the amount of possible states of a system, complexity varies.

KEY WORDS
Software engineering, software size estimation, complexity measure.

1 Introduction

In most classical engineering disciplines it is key to have well-defined models and methods to evaluate, control and manage projects. In order to manage projects, it is necessary to know about the project size. This allows to give statements about resources and optimization during the development process as well as allows the comparison if different implementations and projects [6] [8]. An important challenge in software engineering is the determination of code quantity. During the planning stage of a project, knowing about the complexity of a project allows an adequate project management. It also allows to evaluate alternative implementations. In safety-critical systems, knowing the complexity of single modules enables to rank the modules by their complexity. More complex modules should then be verified in more detail. All these require a software size estimation method, which addresses complexity in an adequate way. For determining projects' sizes key challenge is to control complexity itself. This indicates that complexity is a measure for software size/quantitative analysis of software. The terms are used synonymously.

Compared with classical engineering disciplines which can rely on long-term experience in planning and managing projects, computer science can only rely on a few decades of experience. But also developments and changes happen very fast in computer science. If a project management relies on empirical data from previous projects, transferring these data to current projects is hard and brings high inaccuracies [11]. But today,

However, most cost estimation methods today use empirical data, which are obtained through the analysis of previous projects [10]. All methods have in common, that key figures are used to determine the size of projects and thus make projects comparable. Basically, it is the goal of all these approaches to get complexity under control. Most of these approaches only provide reasonably reliable results for a specific technology, programming language or specification language. But as computer science is subject to rapid changes to technology, programming languages or specification languages, todays complexity estimation methods reach their limitations of reliability very quickly. Thus it would be desirable to have a complexity estimation method that allows for changes in technology and developments, by its very abstract formulation.

One approach is to be independent from design methods and abstraction layers. Therefore, the complexity measurement has to be defined abstractly. Due to this fact, changes in technology, programming languages or specification languages as well as developments could be considered by the measurement itself. If the complexity given by an implementation/realization itself could be measured, comparisons between different projects could be possible.

Measuring software size is historically based on Source Lines of Code (SLOC). This method is often subject of criticism for many reasons [6] [1]. The biggest issues are the high dependency on the style, how code is written and that results from different programming field are not comparable.

Albrecht had presented an approach to use standardized function points [1], which has gained wide popularity to estimate software size [2]. Function points can be applied early on SLOC but it is complicated to get complexity under control [7] [9] [17]. Therefore an objective and comparable measurement of software size is difficult. Even widespread models as COCOMO (COnstructive

COst MOdel) [4] or COCOMO 2.0 [3] are depending on the determination of software complexity.

The paper is organized as follows: the following section will give a brief overview of some of the most used project/software size estimation methods. Afterwards we will introduce the concept of states and the related complexity measure. Section four will then show, how the measurement is applied on software code. This work closes with the conclusion.

2 Complexity measure

Complexity itself needs to be understood to develop an adequate complexity model [5]. In [14] a measurement was proposed which doesn't rely on empirical data. This concept bases its calculations on an abstract variable: states. "A state is a situation in which a system or system's component may be at a certain point of time. It refers to the interior of a system and ignores external influences such as input and output. The set of states is the abstraction of a real system" [15]. This method was used to calculate the complexity of in hardware designs [14] [13]. Thereby, the complexity was not estimated anymore, but calculated [14]. Also, the complexity method has been used to obtain key variables of projects [12]. In [15] the authors of [14] described why their approach with states is capable of addressing complexity in different fields.

The main statements from the works of [14] are summarized in the following in order to apply the method on software in the following. The approach of the design entropy bases on Shannon's information entropy. Shannon's information theory [16], developed by Claude Elwood Shannon, gives mathematical statements about transmitted information. The design entropy concept identifies the single components of a design as sources and drains of information. Connections between components are channels and information are symbols transmitted from a pool of available symbols. In digital hardware connections are normally implemented by wires. The available symbols are "high" and "low" signals in the simplest model. For instance an assignment `a:=b` between two components would be identified as: The information *(=signal level)* from component *(=source/sender)* `b` is transmitted *(=assigned)* to component *(=drain/receiver)* `a`.

The design entropy concept bases its calculations for complexity on states. A system's disorder is a property of a system's state. Therefore the measurement of a system's disorder is complexity. Changing the amount of possible states of a system, complexity varies. "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." [15]. Therefore, states allow to measure project size by the measurement of its entropy. Because states are variables, defined very abstractly. They only depend on the analyzed project's property. It becomes possible to calculate the effect of introducing design tools to a development process by calculating complexity on different abstraction levels and comparing them.

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

Equation (1) shows Shannon's theorem. Comparing the formula to formulas in statistical mechanic, it can be recognized as that of entropy (e.g. [18]). p_α is the likelihood that a system is in cell α of its phase space. H is the same as Boltzmann's H theorem. K is a constant, which "'merely amounts to a choice of unit of measure [16].'" H in formula (1) is the information entropy. Because the mathematical method to calculate complexity is based the idea of information interchange and the formulas can directly be derived from Shannon's information entropy, the complexity measure is also called entropy. Equation (1) can be rewritten as (2) and leads to definition 1 [14] [13].

Definition 1 (Behavior Entropy)**.** *c is a component of a system, with component's sources (inputs) $n_i(c)$, $i = \{1, \ldots, n(c)\}$ and component's drains (outputs) $m_j(c)$, $j = \{1, \ldots, m(c)\}$. The amount of inputs to c is given by $n(c)$. The amount of outputs of c is given by $m(c)$. The amount of possible states for $n_1(c) \ldots n_{n(c)}(c)$ is given by $z(n_i(c))$, $i = \{1, \ldots, n(c)\}$. The amounts of possible states for $m_1(c) \ldots m_{m(c)}(c)$ is given by $z(m_j(c))$, $j = \{1, \ldots, m(c)\}$. $H_B(c) \in \mathbb{R}_0^+$ is the behavior entropy of component c and given by:*

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

The make statements about the complexity to use a component, the behavior entropy is used. One can also speak of a black box or outer look on a component. The behavior entropy does not consider the actual implementation of a component. Therefore the structure entropy has to be used, because this entropy considers the actual implementation of a component by looking at the sub-components used to build up the component. The structure entropy is defined in definition 2. The structure entropy is similar to a white box or an inner view on a component.

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

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

The structure entropy considers a component based approach. With components consisting of sub-components itself, it is necessary to consider the entropy of those sub-components also.

Altogether, the model is derived from fundamental relations. Especially from the information theory, which has proven itself for a long time. Due to the abstract definition of the measurement it can be applied on different abstraction levels. With introducing new tools in development the amount of possible states is influenced. This makes the model interesting for the domain of design automation and project management. New tools and abstraction layers can be considered without the need of a huge amount of experience from previous projects (empirical data). In the following section the measurement will be applied on software.

3 Software size measurement

The following two examples show the usage of the above described method to determine software size. They are reduced to functions, written in C. The first example shows two different implementations for a conditional expression (example 1a and example 1b).

```c
int Comp(int a, int b)
{
    a = (a >= b) ? 100 : 200;
    return a;
}
```

Example 1a

```c
int Comp(int a, int b)
{
    if(a >= b)
        a = 100;
    else
        a = 200;
    return a;
}
```

Example 1b

Looking at the functions, the behavior entropy can be determined by counting inputs, outputs and states of the function. There are two inputs (`int a` and `int b`) and one output (`return a`). There are two possible states for the output (either 100 or 200). Therefore the behavior entropy can be calculated in equation (4) using equation (2).

$$
\begin{aligned}
H_B(\text{Example 1}) &= \log\left(\prod_{i=1}^{n(c)} z(n_i(c)) \cdot \prod_{j=1}^{m(c)} z(m_j(c))\right) \\
&= \log\left(2^2 \cdot 2^1\right) \\
&= \log\left(2^3\right) \\
&= 3 \cdot \log(2)
\end{aligned} \tag{4}
$$

Because the number of inputs and outputs, as well as the number of states is the same for example 1a and 1b, the behavior entropy is the same. As stated before, the behavior entropy does not allow for the actual implementation. In order to determine the complexity of the actual implementation, the structure entropy has to be used. The equation for the structure entropy is given by equation (3).

Therefore, the entropy for the single implementation contents have to be analyzed. The calculation of the structure entropy for example 1a is given in table 1 and for example 2a in table 2. A condition (`if` or `case`) has two inputs, the comparison parameters, and one output (”true” or ”false”) and therefore two states. A `return` statement has one input (the return variable), one output (the return value) and therefore only one state. Lines that are not shown in the tables don't have to be considered because their lack of influence to the complexity calculation.

The result of both calculations is a structure entropy of $H_S(\text{Example 1a/b}) = 10 \cdot \ln(2)$. The result is independent from the actual implementation. It represents the complexity given by the code itself.

In the following, there are two other examples, which show that the measurement can determine the complexity given by the code itself. Therefore we have chosen to compare a `switch` statement in example 2a with an `if – else if` statement in example 2b. To calculate the behavior entropy, equation (2) is used. As in the example before, the behavior entropy has to be the same for example 2a and example 2b, because the number of inputs, outputs and states is the same for the whole component. The components only differ by their (inner) implementation. The behavior entropy of example 2 a/b is then given by equation (5). The components have one input (`int j`), one output (`return`) and four possible states $(0, 10, 100$ and $1000)$. The structure entropy for example 2a and example 2b is given in table 3 and table 4 respectively.

```c
int Log(int j)
{
    switch(j)
    {
        case 1: return 10;
        case 2: return 100;
        case 3: return 1000;
    }
    return 0;
}
```

Example 2a

```c
int Log(int j)
{
    if(j==1)
        return 10;
    else if(j==2)
        return 100;
    else if(j==3)
        return 1000;
    return 0;
}
```

Example 2b

Line	Code	n	m	z	Entropy
3	`(a >= b)`	2	1	2	$1 \cdot 2^2 \cdot \ln(2^2) = 8 \cdot \ln(2)$
	`?  100`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
	`:  200`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
7	`return a`	1	1	2	$1 \cdot 2^1 \cdot \ln(2^1) = 2 \cdot \ln(2)$
					$H_S(\text{Example 1a}) = 10 \cdot \ln(2)$

Table 1. Analysis of example 1a in order to determine its structure entropy

Line	Code	n	m	z	Entropy
3	`if(a >= b)`	2	1	2	$1 \cdot 2^2 \cdot \ln(2^2) = 8 \cdot \ln(2)$
4	`a = 100`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
6	`a = 200`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
7	`return a`	1	1	2	$1 \cdot 2^1 \cdot \ln(2^1) = 2 \cdot \ln(2)$
					$H_S(\text{Example 1b}) = 10 \cdot \ln(2)$

Table 2. Analysis of example 1b in order to determine its structure entropy

$$\begin{aligned} H_B(\text{Example 2}) &= \log\left(\prod_{i=1}^{n(c)} z(n_i(c)) \cdot \prod_{j=1}^{m(c)} z(m_j(c))\right) \\ &= \log\left(4^1 \cdot 4^1\right) \\ &= \log\left(4^2\right) \\ &= 2 \cdot \log(4) \end{aligned} \quad (5)$$

Comparing the results, a higher entropy for the second example was found. Even though example 2b has only two lines of code more than example 1b, the structure entropy is 2.4 times the entropy of example 1. If new lines with braces inside the `if`-statements had been added the entropy wouldn't change, but the lines of code would. Therefore the entropy does not change with the programming style and neither does it with the kind of implementation. It only depends on input, outputs and amount of possible states. This makes not only a comparison of different programmers possible, but also comparing different programming languages. If entropy changes with the language, it is an indication that certain methods are more or less difficult to implement in this language.

4 Conclusion and outlook

In this paper we presented a new approach to determine software size. The used methods, based on states and entropy have already been used to determine complexity of digital circuits. We have shown in this work, that it is possible to apply a abstract measurement on software. In the next step, we will develop an automated analysis for software code, containing a set of rules, how to apply this abstract measurement on written code. Also analysis of complete projects and comparisons with existing software metrics needs to be done.

We consider the here presented approach as a worthwhile starting point for addressing software complexity in an adequate way and for finding a mathematical based method to determine it.

References

[1] A. J. Albrecht. Measuring application development productivity. In I. B. M. Press, editor, *IBM Application Development Symp.*, pages 83–92, October 1979.

[2] A. J. Albrecht and J. E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Trans. Softw. Eng.*, 9(6):639–648, 1983.

[3] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. Cost models for future software life cycle processes: Cocomo 2.0. *Annals of Software Engineering*, 1(1):57–94, December 1995.

[4] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[5] C. N. Calvano and P. John. Systems engineering in an age of complexity: Regular paper. *Syst. Eng.*, 7:25–34, March 2004.

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

Line	Code	n	m	z	Entropy
5	`case 1`	2	1	2	$1 \cdot 2^2 \cdot \ln(2^2) = 8 \cdot \ln(2)$
	`return 10`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
6	`case 2`	2	1	2	$1 \cdot 2^2 \cdot \ln(2^2) = 8 \cdot \ln(2)$
	`return 100`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
7	`case 3`	2	1	2	$1 \cdot 2^2 \cdot \ln(2^2) = 8 \cdot \ln(2)$
	`return 1000`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
9	`return 0`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
					$H_S(\text{Example 2a}) = 24 \cdot \ln(2)$

Table 3. Analysis of example 2a in order to determine its structure entropy

Line	Code	n	m	z	Entropy
3	`if(j == 1)`	2	1	2	$1 \cdot 2^2 \cdot \ln(2^2) = 8 \cdot \ln(2)$
4	`return 10`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
5	`if(j == 2)`	2	1	2	$1 \cdot 2^2 \cdot \ln(2^2) = 8 \cdot \ln(2)$
6	`return 100`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
7	`if(j == 3)`	2	1	2	$1 \cdot 2^2 \cdot \ln(2^2) = 8 \cdot \ln(2)$
8	`return 1000`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
9	`return 0`	1	1	1	$1 \cdot 1^1 \cdot \ln(1^1) = 0$
					$H_S(\text{Example 2b}) = 24 \cdot \ln(2)$

Table 4. Analysis of example 2b in order to determine its structure entropy

[7] N. E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman & Hall, Ltd., London, UK, UK, 1991.

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

[9] T. Hastings. Adapting function points to contemporary software systems: A review of proposals. In *In Proc. 2nd Australian Conference on Software Metrics. Australian Software Metrics Association*, pages 103–114, 1995.

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

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

[12] B. Menhorn and F. Slomka. Entwurfsentropie: Ein Maß im Schaltungsentwurf. In *7th GI/GMM/ITG-Workshop, Multi-Nature-Systems*, 2009.

[13] B. Menhorn and F. Slomka. Project Management Through States. In *IEMS 2009: International Conference on Engineering Management and Service Sciences*, 2009.

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

[15] B. Menhorn and F. Slomka. States and complexity. In *Coping with Complexity COPCOM 2011*, 2011.

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

[17] C. R. Symons. Function point analysis: Difficulties and improvements. *IEEE Trans. Softw. Eng.*, 14(1):2–11, 1988.

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