Abstract- This research investigates software functional complexity, which is loosely defined as the difficulties arising from the functionalities of software, or problem complexity. Software is considered as a task. Therefore, software complexity should be analysed and measured by methods and models of task complexity.

Numerous measurement methods (metrics) have been proposed to measure software complexity, but these have been criticized for their lack of a theoretical model which would serve as a guide for measurement methods. Researchers are now in urgent need of such a model. To fill this need, we propose Wood’s task complexity model in this paper as a theoretical model which will make it possible to both capture and quantify complexity.

Wood’s model analyzes task complexity in three dimensions: component complexity, coordinate complexity and dynamic complexity. We use the first two dimensions of the model to analyze software complexity at an early phase of the software lifecycle (analysis phase). The third dimension is proving to be difficult to capture at this stage, and so has been ignored in our model.

Keywords- Complexity measurement, functional complexity measurement, software complexity, task complexity.

I. INTRODUCTION

Software complexity has been studied for over 25 years now, during which time over 100 different measures have been proposed to capture many different aspects of software complexity (see [13], [15], [29]); however, there is no consensus about what software complexity actually is. What is accepted is that there are two main categories of software complexity: computational and psychological [29].

- Computational complexity refers to algorithm efficiency in terms of the time and memory needed to execute a program.
- Psychological (or cognitive) complexity refers to the human effort needed to perform a software task, or, in other words, the difficulty experienced in understanding or performing such a task.

In the literature, software complexity measures refer to psychological complexity, but there are many different interpretations of their meaning. Zuse (1991) said that “the term complexity measure is a misnomer. The true meaning of the term software complexity is the difficulty to maintain, change and understand software.”

Basili (1980) defined software complexity as “a measure of resources expended by a system (human or other) while interacting with a piece of software to perform a given task.” Henderson-Sellers (1996) challenged this definition, calling it is too broad, and proposed another definition, as follows: “The cognitive complexity of software refers to those characteristics of software that affect the level of resources used by a person performing a given task on it.”

This definition puts emphasis on the software characteristics that generate the level of resources needed to perform the task. Our research perspective adopts this definition. In other words, it focuses on the intrinsic complexity that is derived from the software characteristics and functionalities.

In spite of the fact that many complexity measures have been proposed, software complexity measurement is still in its infancy. The situation in this field is confusing, and not satisfying for the user [29]. Researchers urgently need a theoretical guidance on software measurement in general, and on software complexity measures in particular. Baker et al. (1990) suggested that, “for research results to be meaningful, software measurement must be well grounded in theory.” Kearney (1986) had stated previously that “successful software complexity measure development must be motivated by a theory of programming behaviour”. In this paper, we attempt to apply a cognitive approach to software complexity by using the task complexity model of Wood [28]. First, a software conceptual model is established, and then its elements are mapped to those of Wood’s task model, with the result that software complexity is modeled by Wood’s task complexity model. Finally, some measures are proposed for quantifying software complexity.

Generally, in the literature, software complexity measures capture different aspects of software. Complexity measures are not necessarily appropriate to the software development or maintenance effort expended or to software cost. For example, software complexity in terms of the Cyclomatic Number [20] is related to the number of defects rather than to...
software cost or effort. However, the complexity measures that we are interested in are derived from software functionalities. This approach originated with Function Point methods for software sizing (e.g. Function Point Analysis [4], COSMIC-FFP [1]). Therefore, three of the criteria for a software functional size measure defined in ISO/IEC 14143-1 are adopted here: (1) the measure is applicable at earlier stages of the development process, (2) the measured values are easy to understand from the customers’ perspective, and (3) the measure provides a valid value for estimating efforts and costs. In other words, the complexity measures that are associated with problem are expected as indices of effort, that is, the resources needed to build or maintain software.

This paper, as a first step, is intended only to propose a software complexity model with which functional complexity can be measured. Applying the proposed measures to the estimation of software development effort or software cost is left to future research.

The main contribution of the paper is that it proposes a cognitive approach to software complexity. We argue that software can be seen as a set of functions for accomplishing a task (or tasks) in the real world. This means that it can be modeled as a task. Software activities are also tasks. Therefore, software complexity should be analyzed according to task complexity theory. Wood’s model for task complexity has been selected here because of its concentration on objective complexity, that is, the complexity derived from the characteristics of a task that are independent of the environment in which the task is performed. It is also independent of the person who performs the task. Moreover, this model has been found to be useful in studying software maintenance complexity and software maintenance effort [9].

II. RELATED WORKS

A. Task complexity

In a study by Campbell [5], task complexity was examined along three tracks in the research literature: information processing and decision-making, task and job design and goal-setting. To cut across these research boundaries, task complexity is treated as:

- primarily a psychological experience.
- an interaction between task and human characteristics.
- a function of objective task characteristics.

The complexity associated with the first two approaches is subjective, in that it depends on the capacity of the individual who performs the task. Both these approaches focus on the subjective reactions of an individual to a task, rather than on specific task characteristics. While they might be of great interest in distributing tasks among those performing them, they might be not adequate for objectively measuring the intrinsic complexity of a task. The third approach defines complexity purely in terms of objective task qualities, which is like asking the question, “What makes software complex?”, which this research attempts to answer.

Complexity as objective task characteristics has been studied for decades now. March and Simon (1958) identified some characteristics of tasks which contribute to complexity, such as unknown or uncertain alternatives, and inexact or unknown means-ends connections. Complex tasks are also characterized by the number of subtasks that may or may not be easily factored into nearly independent parts.

Terborg and Miller (1978) defined complexity in terms of path-goal multiplicity. Complexity can arise in two ways. First, the multiple numbers of paths to the goal make it illusory (e.g. solving a jigsaw puzzle), since, although there are many possibilities, only one of them works. Second, there may be several ways to reach a goal, but one needs to choose the best (e.g. finding an optimal path solution for the Traveling Salesman Problem).

Campbell and Gingrich (1986), by studying professional programmers composing computer programs, defined complexity in terms of the interrelated and conflicting elements that place high cognition demands on an individual. They made it clear that “these demands resulted from the nature of the task and not from the characteristics of the individual” [5].

Other research has based task complexity on information cues, which are information units to be treated while a task is performed. For example, Steinmann (1976) suggested that complexity is derived from the amount of information involved in a task, the internal consistency of this information and the diversity of the information itself. Similar concepts have been used by many other researchers to define task complexity (see the survey in [5]).

Campbell also made a distinction between task complexity and task difficulty. He maintained that complex tasks (i.e. of a high degree of complexity) are, by their nature, difficult (i.e. they need a great deal of effort to perform), but that difficult tasks (which require significant effort) are not necessarily complex.

B. Wood’s task complexity model

Wood’s model [28] is generally representative of task complexity. To establish the model, Wood considered four theoretical frameworks: task qua task, task as behaviour requirements, task as behaviour description and task as ability requirements. He regarded the combination of the first two approaches as having the greatest potential for a

2 A task is defined as a pattern of stimuli impinging on the individual.

3 A task is defined in terms of the behavioral responses a person should experience in order to achieve some specified level of performance.

4 Tasks are described and grouped in terms of the kinds of behaviors exhibited when performing the task.

5 Tasks are differential on the basis of the skills required to perform them.
general theoretical analysis of tasks. The model represents task complexity independently of the cognition level of the performer of the task.

Wood’s model proposes three essential elements for task complexity: products, required acts and information cues.

“Products are entities created or produced by behaviours that can be observed and described independently of the behaviours or acts that produce them” ([28], p.64). The task product describes what is expected once the task has been performed. It is independent of acts and information cues, and must be specified before required acts and information cues can be identified. Moreover, product descriptions will include an object plus some defining attributes, such as quantity, quality and timeliness.

An act is as a “pattern of behaviours with some identifiable purpose or direction” ([28], p.65). The required acts (of a task) are actions to perform in order to achieve the goals of the task (i.e. product). An act for the creation of a defined product can be described at any one of several levels of abstraction. It can be a specific activity or a complex pattern of behaviour with an identifiable purpose. Wood considers an act as the basic unit of the behaviour requirement. The direction of an act is the specific kind of activity or process carried out when the act is performed, and so it can be described independently of any individual or the context in which the act is performed.

“Information cues are pieces of information about the attributes of stimulus of objects upon which an individual can base the judgments he or she is required to make during the performance of a task” ([28], p.65).

Using these three elements, task complexity was defined as consisting of component complexity, coordinative complexity and dynamic complexity (see Figure 1). The component complexity ($TC_i$) of a task is a direct "function of the number of distinct acts that need to be executed in performance of the task and the number of distinct information cues that must be processed in the performance of those acts". Wood proposed a formula for this kind of complexity:

$$TC_i = \sum_{j=1}^{n} \sum_{i=1}^{w_i} W_{ij}$$

Where:
- $n$ is the number of acts in the task,
- $w_i$ is the number of information cues to be processed in the performance of the $i$th act of the $j$th subtask,
- $p$ is the number of subtasks.

The coordinative complexity ($TC_2$) of a task takes into account the nature of the relationships between task inputs and task products. In other words, coordinative complexity is the complexity derived from the relationships between the inputs and the products of the task. This relationship can be characterized by the number of acts that need be fulfilled before another act can be performed. As the number of precedence relationships between acts increases, the knowledge and skill required for coordination will increase because the individuals who perform the task will have to learn and perform longer sequences of acts. The formula for capturing this kind of complexity is:

$$TC_2 = \sum_{i=1}^{n} r_i$$

Where:
- $n$ is the number of acts in the task,
- $r_i$ is the number of precedence relations between the $i$th act and all other acts in the task.

Wood also discussed another way of appraising this kind of complexity, which is to examine the form of the relationships between task inputs and task products, i.e. linear or non-linear. To deal with non-linear relationships between information cues and products, an individual will require knowledge of the turning points in the function in order to regulate his or her performance of the task. In this case, the index of coordinative complexity will be the number of turning points that describe the relationships between inputs and products.

**Dynamic complexity ($TC_3$)** is the result of changes in the states of the world that have an effect on the relationships between task inputs and task products. The performance of some acts or the inputs of a particular information cue can generate effects throughout the rest of the task. To quantify this kind of complexity, Wood considers the autocorrelation between component and coordinative complexity from one time period to the next.

Finally, the complexity of the task (or total complexity) is a function of the three kinds of complexity described above. The simplest function for task complexity, proposed by Wood, is a formula representing a linear relationship of component complexity, coordinative complexity and dynamic complexity. He also proposed that the weight of the dynamic complexity be greater than that of coordinative complexity, and then, that the weight of the coordinative complexity be greater than that of component complexity. That is,
Total complexity = $\alpha TC_1 + \beta TC_2 + \gamma TC_3$, where $\alpha < \beta < \gamma$.

III. SOFTWARE FUNCTIONAL MODELS

Our research conforms to the third approach of task complexity (see II.A), i.e., complexity is a function of objective task characteristics. Complexity is defined after software characteristics, and studied independently of the environment in which the software is developed. It is independent of the skill of the programming staff and the tools used. In other words, we try to define and quantify the intrinsic complexity of software or problem complexity.

In this section, we try to establish a software complexity model at the earliest phase of the software lifecycle (analysis phase), which will allow us to study complexity using a task complexity model.

A. COSMIC software functional model

The first challenge is to define “software”. It can be viewed, for example, through a tangible product, such as a document (specifications, system architecture, user guide, etc.), program codes, databases, etc. This can include a nontangible product, such as the functionality it offers to users. Software is also often viewed in several levels of abstraction. At the earliest phases of its lifecycle, software is described in a Software Requirements Specification (also known as User Requirements) of which the functional specifications are a part. Functional specifications describe the “what” to do to respond the user’s needs.

“A Software Requirements Specification is a description of a particular software product, program or set of programs that performs a set of functions in a target environment” (IEEE std 830-1993)

For the abstraction level of software, the generic software model proposed by COSMIC [1] is adopted here. According to this model, software functional requirements can be decomposed into a set of functional processes. Each functional process is a unique set of sub-processes performing either a data movement or a data manipulation (Figure 2).

At a more detailed level, COSMIC distinguishes four types of data movements: Entry, Read, Write and Exit. Each data movement moves only one data group at a time. A data group coming into the software from a user or a device is considered as an Entry. A data group moved out of the software towards the user or a device is considered as an Exit. A data group moved in/out of a storage device is defined as a Write/Read. COSMIC-FFP measures software size by counting the number of data movements, but doesn’t take into account in any way data manipulation [1][27]. Consequently, some modifications are made to this model in the next section to capture the complexity in both data and data manipulation.

B. Proposed software functional model

Essentially, software is built to perform certain tasks in the real world automatically. For a large system, software may be composed of several sub-systems which are independent of one another or weak in relationship to one another, because people often use the divide-and-conquer strategy to control the complexity of a system. Each subsystem can be seen as a set of functions (or “functional processes” in COSMIC-FFP) which work together to fulfill a system task. Each function is a unit act in the real world which is known by the user (or client, i.e. whoever wants to use the software). For example, a banking system may contain subsystems, such as “client account management”, “money exchanges”, etc. The subsystem “client account management” may have some functions: “create monthly transactions report”, “deposit”, “withdraw”, for example. Thus, the software system, subsystem and functions correspond to task, subtask and act in the real world. In addition, each function must be described in a determinate way, including:

- A stimulus (or trigger event), that is, an event such that when it happens, the function is initiated.
- A set of inputs which describes the input information of the function.
- A set of outputs which describes the output information of the function.
- Operation rules, that is, a set of conditions on inputs to obtain different outputs.

A standard specification will help to identify these basic elements without ambiguity.

“A minimum requirement is that the software requirement specifications provide a description of every input (stimulus) into the system, every output (response) and all functions

\[ \text{Total complexity } = \alpha TC_1 + \beta TC_2 + \gamma TC_3, \text{ where } \alpha < \beta < \gamma \]

\[ \text{A. COSMIC software functional model} \]

\[ \text{B. Proposed software functional model} \]

\[ \text{III. SOFTWARE FUNCTIONAL MODELS} \]

\[ \text{At a more detailed level, COSMIC distinguishes four types of data movements: Entry, Read, Write and Exit. Each data movement moves only one data group at a time. A data group coming into the software from a user or a device is considered as an Entry. A data group moved out of the software towards the user or a device is considered as an Exit. A data group moved in/out of a storage device is defined as a Write/Read. COSMIC-FFP measures software size by counting the number of data movements, but doesn’t take into account in any way data manipulation [1][27]. Consequently, some modifications are made to this model in the next section to capture the complexity in both data and data manipulation.} \]

\[ \text{A standard specification will help to identify these basic elements without ambiguity.} \]

\[ \text{A minimum requirement is that the software requirement specifications provide a description of every input (stimulus) into the system, every output (response) and all functions} \]

\[ \text{In [28], it is written as } \alpha < \beta < \gamma. \text{ There is an incoherence between the explanation in formula and that in the written description.} \]

\[ \text{The COSMIC-FFP Measurement Manual ([1] p. 15, 22, 49) points out that the COSMIC-FFP method was not designed to take into account complex algorithms. Therefore, the method is only suitable for sizing “movement-rich” types of software.} \]

\[ \text{Our previous study [27] attempted to capture the complexity in data manipulation. Data manipulation was interpreted in terms of relationships between inputs and outputs rather than algorithms, because it seems to be impossible to get enough detail on algorithms from the software requirement specifications.} \]
performed by the system either in response to input or in support of an input” (IEEE std 830-1993).

From the above examination, a generic model for software in the analysis phase is proposed in Figure 3. Software is a set of functions, each of which has four basic elements: stimulus, inputs, outputs and operations rules.

Each input/output of a function is a piece of information (i.e. termed by Wood an "information cue") which comes into/out of the function. A piece of information in our model is equivalent to a data group in the COSMIC model. However, we identify only the information cue rather than the four COSMIC function types. In practice, there is not much difference between the number of data movements and the number of information cues, because each data movement is related to one data group, and, very often, in a functional process, a data group is related to only one data movement. If each data group in the COSMIC model is related to only one movement, then the number of data movements will be the number of information cues. Using this construct, an input/output is now naturally mapped to an information cue of Wood's model. Table 1 represents the mapping between the elements of our software model and those of the Wood task complexity model.

<table>
<thead>
<tr>
<th>Software elements</th>
<th>Wood elements</th>
<th>model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software system</td>
<td>Task</td>
<td></td>
</tr>
<tr>
<td>Subsystem</td>
<td>Subtask</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Act</td>
<td></td>
</tr>
<tr>
<td>Input, output data group</td>
<td>Information cue</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Mapping between software model elements and Wood's task model elements.

Operation rules in a function represent relationships between inputs and outputs. They are characterized by the conditions placed on inputs to obtain different outputs. This can be considered as a modification of the COSMIC model to deal with the complexity in data manipulation. Therefore, the software functional model proposed here captures not only input and output data, but also data manipulation, which processes inputs to produce outputs.

IV. SOFTWARE COMPLEXITY MODEL

Now, software is seen as a task, and its elements correspond to the elements of the task. It is clear that software complexity should naturally be investigated from the point of view of task complexity. The first two dimensions of Wood’s complexity model, i.e. component complexity and coordinative complexity, are adapted for software complexity. The third dimension in the Wood model, dynamic complexity, is ignored, because, as defined by Wood, this kind of complexity varies over time. However, it is supposed that each software function, when it is defined, is deterministic and invariant.

In previous research [26], through an analysis of the way software size is measured with FPA (Function Point Analysis [4]), it was concluded that, in addition to measuring software size, FPA also takes into account software complexity, and the fact that there are two categories of complexity: component complexity and system complexity.

Component complexity refers to the internal complexity of a functional process which is characterized by the input data and output data, as well as the data manipulation that processes inputs to produce outputs.

System complexity refers to the complexity in the relationships between functional processes. This kind of complexity is characterized by the data communication between functional processes.

IV. COMPLEXITY MEASURES

From the conceptual model in Figure 4, three software complexity measures are proposed:

- The first measure is the number of data groups (NOD). This number is essentially the number of information cues, and therefore this measure totally corresponds to Wood’s component complexity. It is also similar to software size as measured by the COSMIC-FFP method, but measures by
These three measures are aimed at capturing three aspects of complexity in software: the number of conditions (NOC) and the entropy of system functions.

The second measure is the number of conditions (NOC) on inputs required to produce different outputs. It aims to capture the relationships between inputs and outputs. It deals with data manipulation, which was ignored in the COSMIC-FFP method. This measure is essentially the number of turning points in Wood’s task complexity model. It is also similar to McCabe’s cyclomatic complexity [20]. The complexity in data manipulation, measured by NOC, is interpreted as the complexity in the relationships between inputs and outputs expected rather than algorithmic complexity as proposed by COSMIC. This measure aims to deal with the fact that two functional processes with the same input and output data groups can be different in terms of data manipulation. However, we focus on the conditions to obtain outputs rather than on algorithms to produce outputs because the specifications of the algorithms are not common in software requirement specifications.

In the literature, Cyclomatic complexity is accepted as a good measure of the complexity of program structure. This measure is strongly related to the number of defects and difficulties encountered in testing and maintenance software [21]. Cyclomatic complexity is defined based on a graph which represents the flowchart of the program. It can be seen as the number of condition nodes (in the flowchart) plus one, and is interpreted as the number of independent paths of the flowchart. By analogy, NOC captures complexity in the “structure” of the functional process.

The third measure takes into account the relationships between functions (or functional processes). In our previous research [26], this complexity is characterized by the data groups in communications between two functions (see Figure 4). Although based on the same idea as the Wood model, that is, it is the complexity between functions that should be taken into account; our model captures this kind of complexity in a way that differs from Wood’s model. In fact, the relationship of one act to others is characterized by the number of previous acts which must be accomplished before this one is considered in Wood’s model. In the software context, this kind of complexity is analysed by the data groups in communications between two functions, rather than by the number of functions that must be fulfilled before a particular function is considered. This means that data coupling between functions is captured, rather than the order of execution of functions. In the following section, an Entropy (EOS) measure will be proposed for quantifying the complexity derived from data communications between functions.

In summary, the number of data groups (NOD), the number of conditions (NOC) and the entropy of system (EOS) are three measures proposed for software complexity. These three measures are aimed at capturing three aspects of complexity in our conceptual model (Figure 4), which conforms well to Wood’s task complexity model.

VI. ENTROPY MEASURE FOR SYSTEM COMPLEXITY

The notion of Entropy is used as an information measure in the theory of communication proposed by Shannon. Davis and LeBlanc (1988) also developed an entropy-based software complexity measure. According to their model, the basic element in a program is a chunk of code. A chunk can be a single statement or a block of code, or even a module which is a “logic segment” of a program. They argue that programmers do not understand a program statement by statement. Rather, they associate groups of statements that have common functions, called chunks. Program understanding, therefore, is based on the semantic links between chunks. The semantic links are characterized by “control” between chunks (i.e. a chunk may transfer controls out of the chunk), and data dependency between chunks (i.e. a chunk has a variable, the value of which is modified in another chunk). Davis and LeBlanc proposed an entropy measure to capture the complexity of the semantic links between chunks as follows:

A program is seen as a directed graph representing semantic links between chunks. A node is a chunk. A directed arc represents a control from one chunk to another chunk or a data dependency.

Chunks are classified into chunk-equivalent classes. Two chunks are equivalent if and only if they have the same in-degree and out-degree, i.e. they have the same number of in-arcs and out-arcs. For example, the nodes in graph G are classified in four equivalent classes \{a\}, \{b,c,e,f\}, \{d\}, \{g\}.

Entropy is then defined by the frequency of appearance of chunks in an equivalent class by the formula:

\[ H = - \sum_{i=1}^{n} p(A_i) \log p(A_i), \]

where \( n \) is the number of equivalent classes; and \( p(A_i) \) is the possibility of a chunk falling into the \( i^{th} \) class. For example, the entropy of graph G is:

\[ H = -[1/7 \log(1/7)+ 4/7 \log(4/7) + 1/7 \log(1/7) + 1/7 \log(1/7)] = 1.664. \]

The minimum entropy occurs when all chunks are in the same class. In this case, \( n=1 \), \( p(A_1) = 1 \), therefore \( H = 0 \).
The maximum entropy occurs when every chunk is unique, i.e. each class has only one chunk. In this case, 
\[
H = \sum_{i=1}^{n} \frac{1}{n} \log \frac{1}{n} = \log n
\]

In their research, Davis and LeBlanc also distinguished between data dependency and data control. They found that:
- Data connections among chunks are more significant than control connections in predicting debugging time and number of errors.
- Control connections among chunks tend to be more significant than data connections for predicting construction time.

Motivated by the results of Davis and LeBlanc, an entropy measure is proposed to capture the complexity in relationships between functions. No distinction is made between data control and data dependency; instead, the term data communication is used. Two functions have a data communication if an output of one function becomes an input of the other. If one function sends a signal to trigger another function without sending data to process or receiving a feedback result, then there is no data communication. In other words, the trigger signal itself is not a data communication.

A directed graph is used to represent the relationships between software functions. A node on the graph is a function. An arc represents a data group in a communication between two functions. Therefore, there may be more than one arc from a node A to a node B because there are many data groups going out of A and coming into B. To simplify the graph, weighted arcs are used, as in Figure 6. Each arc in the graph is assigned a weight representing the number of data groups in the communication.

To calculate entropy, software functions are also classified in equivalent classes. Two functions belong to the same equivalent class if and only if they have the same in-degree and out-degree, that is, the sum of the weight of arcs coming into and out of the function, respectively. For example, graph G’ in Figure 6 has the following equivalent classes:

\[ \{a\} \text{ with in-degree } = 0 \text{ and out-degree } = 3 \]
\[ \{b,f\} \text{ with in-degree } = 1 \text{ and out-degree } = 2 \]
\[ \{c,e\} \text{ with in-degree } = 2 \text{ and out-degree } = 3 \]
\[ \{d\} \text{ with in-degree } = 4 \text{ and out-degree } = 2 \]
\[ \{g\} \text{ with in-degree } = 5 \text{ and out-degree } = 0. \]

The entropy of data communication in graph G’ is
\[
H = \frac{1}{7} \log \frac{1}{7} + \frac{2}{7} \log \frac{2}{7} + \frac{2}{7} \log \frac{2}{7} + \frac{1}{7} \log \frac{1}{7} + \frac{1}{7} \log \frac{1}{7} = 2.24.
\]

VII. EXAMPLE

The proposed model is now applied to measurement of the complexity of an application which simulates a Rice Cooker. This is a small case study to show how to apply the model. The COSMIC-FFP functional size of this application is also compared with the number of data groups our model has. Here, the specification of the Rice Cooker is based on the one published in a case study of the Software Engineering Management Research Laboratory. For the purposes of this paper, some modifications have been made. The following specification can be seen as a simplified version of the specification of the Rice Cooker in ([18], p.18-19).

A. Specifications of the Rice Cooker

The interface of the Rice Cooker is illustrated in Figure 7.

\[ \text{Figure 7: Operation panel of a Rice Cooker [18].} \]

The Rice Cooker must be able to cook rice in three modes: fast, normal and gruel.
- The Rice Cooker starts cooking the rice when the Start button is pressed, normally after a mode has been selected. If no mode is selected, the Rice Cooker operates in normal mode by default.
- Once the rice is completely cooked, the Rice Cooker enters warming status.
- The appropriate indicator lamps must be lit during...
cooking or warming to indicate Rice Cooker status.

- The heater is controlled according to the cooking mode and to the actual temperature.
- The target temperature of the heater for a given mode at a given elapsed time is predefined, as in Figure 8. For every 30 seconds, a new target temperature will be set, and
- For every 5 seconds, the actual temperature will be obtained (from a sensor) and compared with the target temperature. If the target temperature is higher than the actual temperature, the heater must be in ON status, otherwise it must be in OFF status.

B. Complexity of Rice Cooker application

From the specification, four functional processes are identified in the cycles in Figure 9. The complexity measured by the number of data groups (NOD) appears in Table 2. A detailed comparison with COSMIC-FFP functional size\(^8\) (Cfsu) is also made here (note that E, W, R and X indicate Entry, Write, Read and Exit).

![Figure 9: Functional processes & data communication between processes in the Rice Cooker application.](image)

Note that, in the Select cooking mode process, Cooking mode is counted only once, but it is counted twice with the COSMIC-FFP method. Again, in the Control elapsed time process, Elapsed time is counted once, while it is counted twice with the COSMIC-FFP method. That is why the COSMIC-FFP total is slightly higher than the NOD total.

Complexity in data manipulation, measured by the number of conditions, is as follows:

- For the Select cooking mode process: NOC = 0
- For the Set target temperature process: NOC = 1, because there is one condition:
  - IF (Elapsed time < Cooking time)  
  - Else Cooker status = OFF
- For the Control Heater process: NOC = 1, because there is one condition:
  - IF (Actual temperature < Target temperature)  
  - Heater status = ON

\(^8\) Functional size is measured using the COSMIC-FFP Measurement Manual [1], and is not the result published in [18].

<table>
<thead>
<tr>
<th>#</th>
<th>Process</th>
<th>Trigger event</th>
<th>Information cue</th>
<th>Cfsu</th>
</tr>
</thead>
</table>
| 1 | Select cooking mode | Button pressed | **Cooking mode** (NOD = 1) | E: Cooking mode  
W: Cooking mode |
| 2 | Set target temperature | 30s signal event | **Elapsed time**, **Cooking mode**, **Predefined cooking time**, **Predefined temperature data**, **Target temperature**, **Cooker status** (NOD = 6) | E: Elapsed time  
R: Cooking mode  
R: Predefined cooking time  
R: Predefined temperature data  
W: Target temperature  
X: Cooker status, Cfsu = 6 |
| 3 | Control Heater | 5s signal event | **Target temperature**, **Actual temperature**, **Heater status On/Off** (NOD = 3) | R: Target temperature  
E: actual temperature  
X: set heater On/Off, Cfsu = 3 |
| 4 | Control Elapsed time | Start button pressed | **Clock signal**, **Elapsed time**, **30s event**, **5s event** (NOD = 4) | E: Clock signal  
W: Elapsed time  
X: Elapsed time  
X: 30s event  
X: 5s event |

| | Total NOD = 14 | Total Cfsu = 16 |

Table 2: Number of data groups of functional processes.

Data communication between the functional processes in the Rice Cooker application is represented by arrows in Figure 9. Four functional processes are classified in three equivalent classes:

- [Select cooking mode, Control elapsed time] with in-degree = 0, out-degree = 1.
- For **Control elapsed time**, 5s and 30s signals are not counted as data groups in communication, because they are trigger events; **Elapsed time** is the only data group in communication.
- [Set target temperature] with in-degree=2, out-degree= 1.
- [Control Heater] with in-degree = 1, out-degree = 0.

Therefore, the system complexity of the Rice Cooker is:

\[
\text{EOS} = - \frac{2}{4} \log(2/4) + \frac{1}{4} \log(1/4) + \frac{1}{4} \log(1/4) = 1.5.
\]
The complexity of the Rice Cooker application is now represented as a vector of three dimensions (14, 4, 1.5).

VIII. CONCLUSION

This paper investigates a task complexity model with the idea of establishing some measures for software complexity. The conceptual model is defined using the Wood task complexity model. The main contribution of this work is to build a software complexity model based on a theoretical cognitive model of the task. The proposed measures are guided by task complexity theory.

The proposed measures have not yet been experimentally validated, but they intuitively conform to some measures which are already known. NOC is similar to McCabe's cyclometric complexity, but applied at a higher abstraction level. System entropy (EOS) is a new type of measure. In software metrics, there are some methods that measure the complexity of code by the entropy of symbols in the program [14][11], but there are not many methods that apply the notion of entropy at a higher abstraction level (i.e. before codes are done).

There remain some pertinent research questions to consider:

How do these measures contribute to an effort/cost model? How can these measures be used to explain the level of difficulty of some software activities, system integration or system test, for example? Can the three proposed measures be used to accurately estimate software effort/cost?

REFERENCES


