CompSize: Automated Size Estimation of Embedded Software Components

Kenneth Lind
Electrical Systems Engineering
Saab Automobile AB
Trollhättan, Sweden
e-mail: kenneth.h.lind@saab.com

Rogardt Heldal
Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
e-mail: heldal@chalmers.se

Tigran Harutyunyan
University of Gothenburg
Gothenburg, Sweden
e-mail: tikogrig@gmail.com

Tony Heimdahl
Chalmers University of Technology
Gothenburg, Sweden
e-mail: tony.heimdahl@gmail.com

***Abstract*—Accurate estimation of Software Code Size is important for developing cost-efficient embedded systems. The Code Size affects the amount of system resources needed, like ROM and RAM memory, and processing capacity. In our previous work, we have estimated the Code Size based on CFP (COSMIC Function Points) within 15% accuracy, with the purpose of deciding how much ROM memory to fit into products with high cost pressure. Central in that work is the mapping between CFP and the information available early in the development process. We have previously defined a UML Profile capturing the information needed for CFP measurement and estimation of Code Size. The key idea was to extend UML components to contain all the necessary information. In this paper, we show how we developed a tool for automated estimation of Code Size based on our UML Profile. The tool is designed to permit Code Size estimation based on other UML diagrams than components. A case study evaluates the UML Profile and the tool in a realistic case.**

Keywords-software component; functional size measurement; code size estimation; UML Profile

I. INTRODUCTION

Early and accurate estimation of Software Code Size is important for developing cost-efficient embedded systems, such as cars, cell phones, washing machines, etc. The Code Size affects the amount of system resources needed, like ROM and RAM memory, and processing capacity. Systems containing too much memory or processing capacity are more expensive than they need to be. Systems containing too little memory or processing capacity may need a redesign after only a part of its expected lifetime.

In our previous work, we have estimated the Code Size based on CFP (COSMIC Function Points) within 15% accuracy [23],[24],[25],[26],[27],[28]. Our results were obtained using software implementations developed by the automotive companies Saab and GM (General Motors). The accuracy of the estimated values is important because the purpose was to decide how much ROM memory to fit into ECUs (Electronic Control Unit, an embedded computer) in products with high cost pressure. We investigated requirement specifications and UML models available early in the development process and identified a 1-to-1 mapping between the available information and the COSMIC method. Finally, we defined a UML Profile capturing all information needed for CFP measurement and estimation of Code Size.

In this paper, we present the CompSize tool that we have developed, and that is based on the mapping rules we have established in our previous work. We show that the CompSize tool can import the information modeled using our UML Profile to achieve automated estimation of Code Size based on CFP. Besides the increased efficiency obtained by our automated estimation approach [29], we expect to increase repeatability and consistency in the estimation process compared to a manual approach. We conduct a case study using requirement specifications and software implementations from the automotive industry to evaluate the tool.

This paper is organized as follows: The next section provides background information about the COSMIC method and our previous work. Section 3 presents the UML Profile, and section 4 describes the tool. Section 5 explains the case study, section 6 contains related work, and finally section 7 reports on conclusions.

II. BACKGROUND

This section presents enough information about Functional Size Measurement and the COSMIC method to understand the rest of the paper. It also briefly explains our domain and parts of our previous work that are important for this paper.

A. Functional Size Measurement

Functional Size is defined as "size of the software derived by quantifying the Functional User Requirements" [5]. FUR (Functional User Requirement) describes what the software is expected to do for its users. Examples are data transfer, data transformation, data storage, and data retrieval. Functional Size is independent of software language and development methods.

There are several FSM methods available. A comprehensive literature survey covering several methods is found in [9]. The typical usage of FSM is development cost estimation and project planning. In our work, COSMIC Function Points (CFP) [19] is chosen because it is known to be suitable for real-time software, like automotive systems [5], and it is a "second generation" method, complying with the ISO/IEC 14143-1:2007 standard for FSM methods [13],[14],[15],[16],[17],[18].

The COSMIC Method defines a standardized measure of software Functional Size expressed in CFP units. The measurement is carried out by mapping the FUR of the software onto the COSMIC Generic Software Model (shown in Figure 1). The purpose of the measurement and scope of the software to be measured defines the level of decomposition and level of granularity of the software. The level of decomposition points out a particular level in a software – component – sub-component hierarchy. The level of granularity concerns the amount of details defined about the FUR. Both aspects are important when comparing different CFP values to each-other.

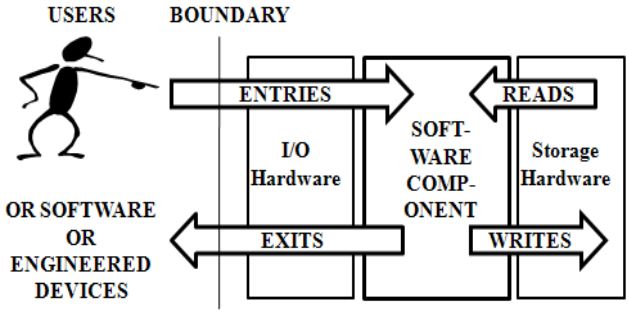

Figure 1. The Generic Software Model of COSMIC.

As can be seen in Figure 1, there are four different data movement types. Entry types move data across the boundary and into the functional process. Exit types move data across the boundary to a user. Read types move data from persistent storage to the functional process. Write types move data from the functional process to persistent storage. Persistent storage (Storage Hardware in Figure 1) enables a functional process to store data from one execution cycle to another. Each data movement is equivalent to 1 CFP, and operates on a common set of attributes.

B. Our domain and previous work

UML Components [31],[35] are often used to model complex systems by decomposing a large system into smaller parts. Saab and GM uses UML Component Diagrams to show how the customer feature is divided into its smallest entities called "distributable components", and the interfaces between them. A distributable component must never be split up into more components, but can be used in several features. The UML Component Diagram is modeled in the IBM Rational Rhapsody tool [11], as part of the system architecture development activities within Saab and GM. This is described further in [4].

In our previous work, we have used existing distributable components of the type shown in Figure 2. In this diagram, we see that the distributable components are modeled as component stereotypes denoted "Distributable" followed by the name of the component. As we can see from the diagram, the Truck Bed Cargo Lamp component has three required interfaces (arrows going in to the component) and one provided interface (arrow going out from the component). The Component Diagram deviates from standard UML for reasons that are explained in [25].

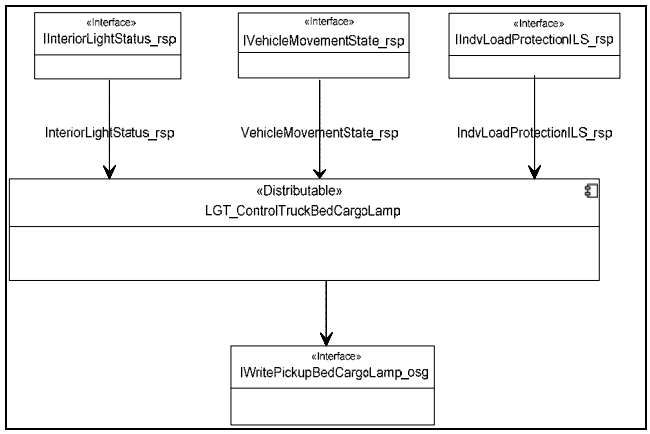

Figure 2. Component Diagram of the Truck Bed Cargo Lamp component.

The Component Diagrams do not contain all the information we need to measure the Functional Size. We also need the requirement specifications related to the components. In the requirement specification we find in textual form the information needed such as: calibration parameters (used for tuning of a general software component to a certain type of product), persistent storage of variables in RAM-type memory, etc. The textual requirements for the software component in Figure 2 are shown in Figure 3.

The feature **Shall** be enabled when the calibration CARGO LAMP PRESENT is set true. <END>

If the vehicle power mode is "OFF", and the cargo lights are illuminated, the SYSTEM **Shall** keep the cargo lamps active as long as Inadvertent Load Control power is active. <END>

CUSTOMER "ACTION"	CUSTOMER PERCEIVABLE "OUTPUT"	MAXIMUM LATENCY "ACTION" to "OUTPUT"
INTERIOR ILLUMINATION Lamps Switch On and Vehicle Parked.	***Cargo Lamp Illuminates***	**100 ms**

Figure 3. Extract from a requirement specification for the Truck Bed Cargo Lamp component.

The mapping between the component specified in Figure 2 and Figure 3 is shown in Figure 4.

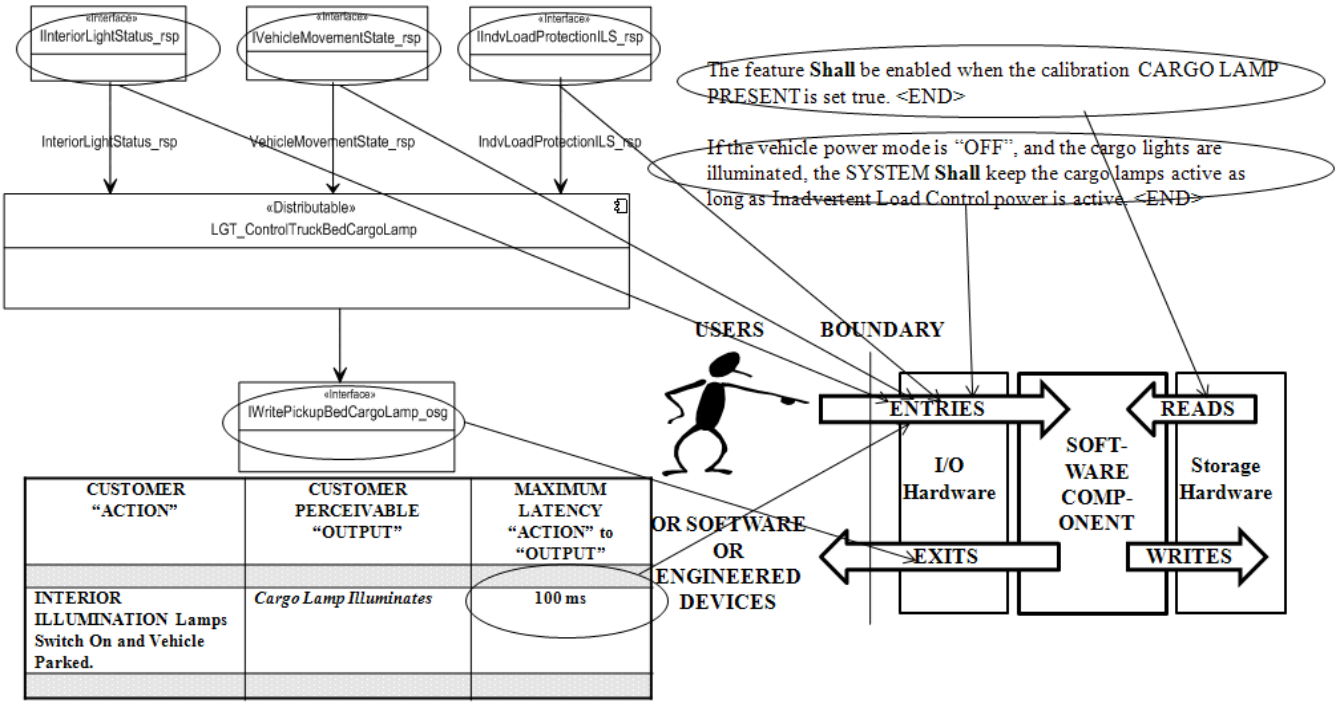

Figure 4. Mapping of a distributable component onto the COSMIC Generic Software Model.

In Figure 4 we see that the required interfaces are measured as Entry data movements, and the provided interface is measured as an Exit data movement. The maximum latency requirement in Figure 3 is measured as an Entry data movement (a Clock tic), because it will be implemented as a periodic invocation of the component. The vehicle power mode requirement in Figure 3 is in fact a required interface, and it is measured as an Entry data movement. The CARGO LAMP PRESENT requirement in Figure 3 is measured as a Read data movement. The result is CFP=7, i.e. 5 Entry data movements + 1 Exit data movement + 1 Read data movement.

We have used this mapping to measure CFP for 46 components of 4 different types so far. We had access to the implementations of the same components, from which we obtained the Code Size in Bytes. For each type of component we established strong correlation between CFP and Bytes, which enabled us to design linear models using linear regression on the measurement data [23],[24],[25],[28]. Our studies have recently been replicated by Renault using their own data, and they published similar results [34].

By investigating our measurement data [27], we have identified factors to use for categorization of software in our domain, in order to select the proper linear model for estimation of Code Size. The categorization is important to increase the estimation accuracy, by using historical data from implementations of similar software. This way we can capture algorithmic complexity and manipulation of large amounts of data, although COSMIC cannot measure this directly.

To summarize this section, we conclude that the main concepts we need to consider in COSMIC are the Generic Software Model (containing users, boundary, functional processes, and data movement types), the level of decomposition, and the level of granularity. In addition, categorization of software is important for estimation of Code Size. How these concepts can be modeled in UML will be described in the next section.

III. A UML Profile for Code Size Estimation based on COSMIC

In previous work we have defined a UML Profile capturing all information needed for CFP measurement and estimation of Code Size [29]. Because it is an important part of our automated estimation approach, we will briefly present the usage of the Profile.

Our goal is to define how to model the COSMIC Generic Software Model using UML. UML components have a natural boundary between the software and its users, in a similar way as in COSMIC. Therefore we base our UML Profile on UML components. We will explain how to use the UML Profile by describing how to model the component specified in Figure 2 and Figure 3, see Figure 5. The interfaces in Figure 2 are modeled using standard UML notation in Figure 5, to show that our tool supports standard UML. In Figure 5 we see that required interfaces are related to the component by a <<use>> dependency, and provided interfaces are related to the component by a realization relationship.

The textual requirements such as the ones in Figure 3 are the information we need to model using the UML Profile. In our work we use the term "parameter" to represent a variable within a component which can be read, written, or read/written from/to memory. One such parameter is the

CARGO LAMP PRESENT requirement in Figure 3, and therefore it is modeled as a stereotype Parameter with the value of Direction assigned to "in" in a class called Variables. The value "in" in the example represents Read data movement. Other possible values are "out" that represents Write data movement, and "inout" that represents a combined Read/Write data movement.

The maximum latency requirement in Figure 3 is modeled as a required interface, because it will be implemented as a periodic invocation of the component. The vehicle power mode requirement in Figure 3 is a required interface, and it is modeled accordingly. So in total we have 5 required interfaces and 1 provided interface.

This far we have captured the needed information for COSMIC in UML using our Profile, and we can define mapping rules as support for calculating the CFP value. These mapping rules are shown in Table 1. Using the rules defined in Table 1 to calculate the CFP value, the result is CFP=7, i.e. 5 Entry data movements + 1 Exit data movement + 1 Read data movement.

In addition to the Entry, Exit, Read, and Write data movements, we need to model the values of the Categorization factors needed for estimation of Code Size. To do this we use the stereotype Category and assign values to its attributes named functionality, constraints, etc. Figure 5 shows that the attribute values assigned are "Comf & Conv" to functionality, "no" to constraints, etc.

The model in Figure 5 illustrates an example of how to use our UML Profile defined in [29], and the mapping rules between COSMIC and our UML Profile are shown in Table 1.

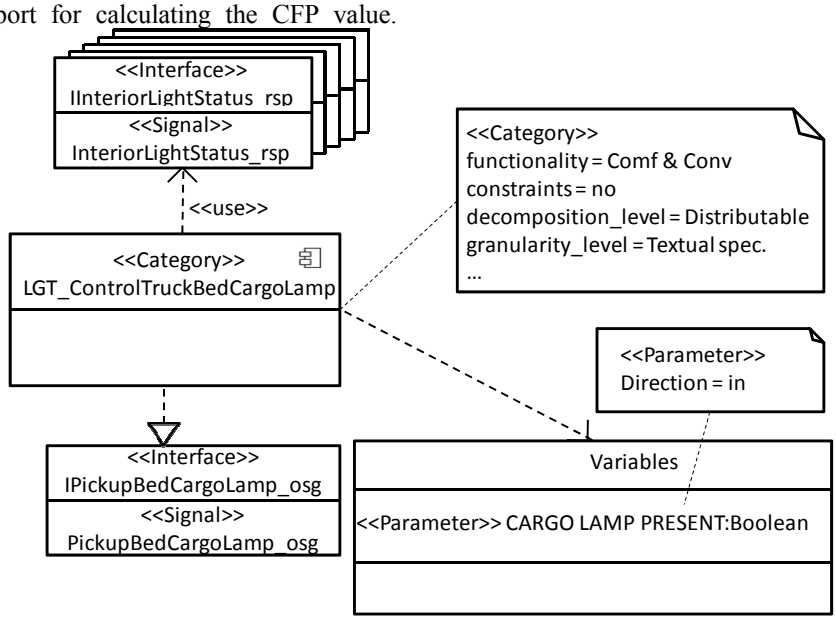

Figure 5. Mapping of a distributable component onto the UML Profile.

TABLE 1. MAPPING RULES BETWEEN MAIN COSMIC CONCEPTS AND THE UML PROFILE.

COSMIC concept	UML concept
Functional process	The functional requirements contained in the component. Must reside completely within one component.
User	Surrounding components.
Boundary	Component boundary.
Level of granularity	Part of categorization.
Level of decomposition	Part of categorization.
Entry data movement	Operation in required interface.
Exit data movement	Operation in provided interface.
Read data movement	Parameter with direction=in.
Write data movement	Parameter with direction=out.
Read/Write data movement	Parameter with direction=inout.

The main purpose of the component diagrams within Saab and GM is to show the components with their interfaces, and not to estimate implemented Code Size. The model in Figure 5 illustrates an example of how to use our UML Profile. The Profile uses notes to give values to stereotype attributes, which is the suggested way by the UML specification [31],[32]. The problem with this representation is that the models become cluttered with information only needed for Code Size estimation, and becomes harder to understand for other purposes. In addition, using notes to define stereotype attribute values requires extra modeling effort. We want to avoid the cluttering of the graphical models, as well as minimize the modeling effort. Therefore we have tailored a representation of our UML Profile to the Rhapsody modeling tool currently in use at Saab and GM. For completeness we will describe this representation next. We will show how we capture the model in Figure 5 in the Rhapsody modeling tool, see Figure 6.

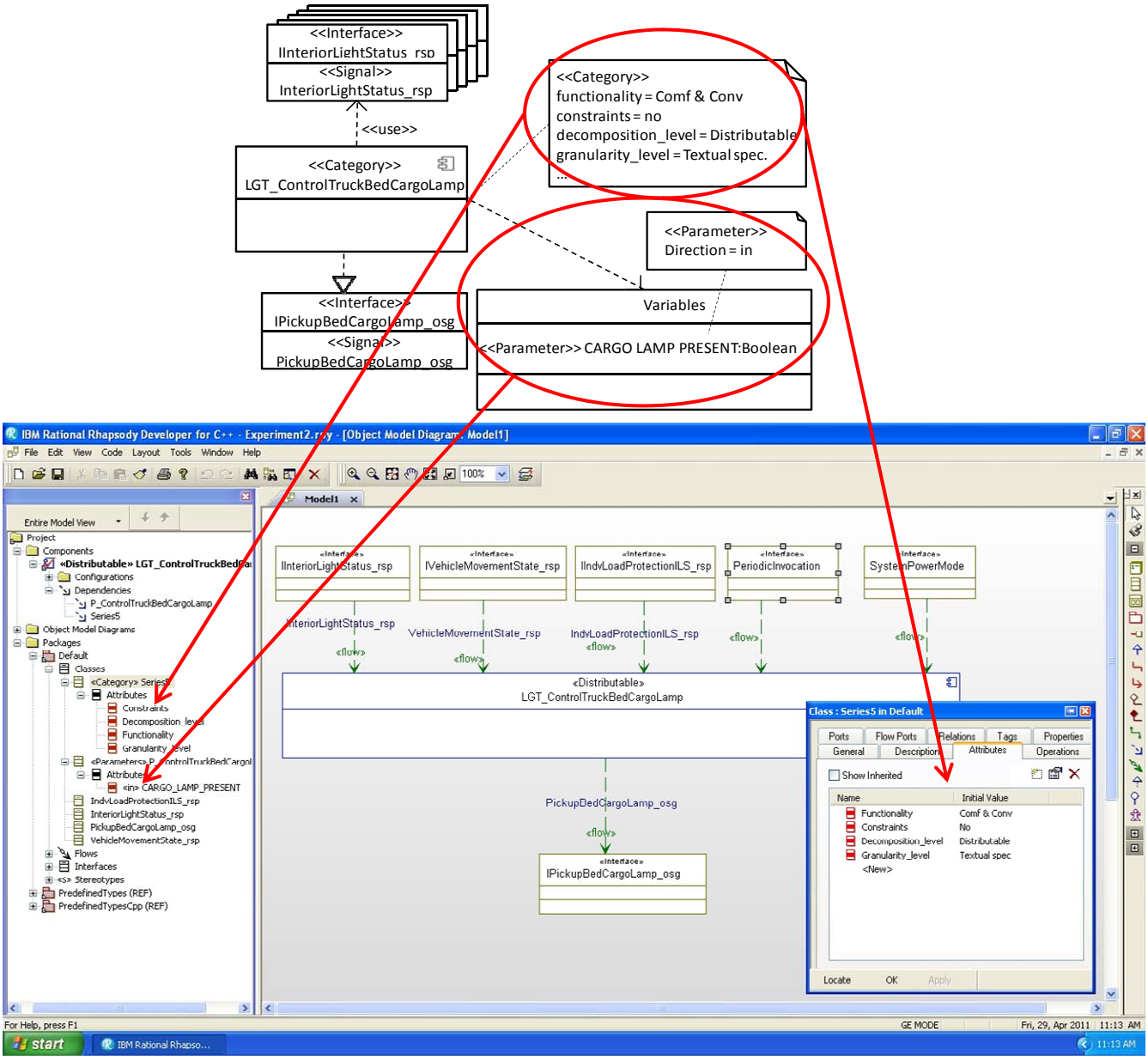

Figure 6. The UML Profile modeled in Rhapsody.

The UML model shown in Figure 6 contains the same information as the model in Figure 5, but with a different notation. In the Rhapsody model we also represent the parameters of a component with a class. In Rhapsody this class is marked by the stereotype <<Parameters>> and a dependency between the class and the component. If there are several parameters, they will show as several attributes to the class. The information shown in the stereotype <<Parameter>> in Figure 5 about the direction can be given directly to attributes in Rhapsody, as is shown in Figure 6. Here the attribute CARGO_LAMP_PRESENT is given the direction <<in>>, because data is read from memory into the component. The other two possibilities are <<out>> and <<inout>>. The information is included in the tree structure of the model but not in the graphical representation of the model. This reduces the cluttering of the graphical model and simplifies the modeling work, compared to the notation in Figure 5.

In Figure 5, category is given as a note. In Rhapsody we represent the same information using a class with a stereotype <<Category>> and a dependency between the class and the component. The attributes of the class represent the factors of the category with values. In Figure 6 we see that the Series5 class has attributes Constraints, Decompositon_level, Functionality, and Granularity_level. These attributes are given values using the "Class" window shown to the right in Figure 6. Here we see that Functionality is assigned the value "Comf & Conv", Constraints is assigned the value "No", Decomposition_level is assigned the value "Distributable", and Granularity_level is assigned

the value “Textual spec”. Again, the information is included in the tree structure of the model but not in the graphical representation of the model. Another benefit is that a <<Category>> class that is already defined can be reused for other components with the same category.

In [29] we defined a UML Profile that captures the information needed for CFP measurement and Code Size estimation. In this section we have illustrated an example of how to use this Profile. We have also explained how the same information can be modeled using Rhapsody, without cluttering the component diagrams and with minimum modeling effort. The UML Profile will be evaluated in a case study later in this paper. The mapping rules summarized in Table 1 was implemented in a tool, which will be described in the next section.

IV. THE COMPSIZE TOOL

We have developed a tool based on our results from 3 years of research. The tool was implemented in Java JDK 1.6 using Eclipse IDE for Java Developers resulting in around 1,7 MBytes of code, and required 6 man months of effort.

The main functionalities of the tool are; to import information modeled using the UML Profile described in the previous section, to store component data needed for CFP measurement and Code Size estimation, to calculate estimated Code Size using linear regression, and to present estimation results.

The details of the tool will be presented in this section.

A. CompSize tool overview

Here we will explain the main features of the tool. The main window of the tool is shown in Figure 7. All the information about the components and categorization can be found in the Components Display tab in the upper section of the window. Here you find the name of the component, the number of Entry, Exit, Read, Write data movements, the CFP value, estimated and real code Size in Bytes, etc. The DFP value in column 6 is defined in [28] as a measure that only counts the Entry and Exit data movements. The tool also supports calculation of DataSize (RAM memory size) and estimation of development effort. For that purpose it can store estimated and real DataSize, as well as estimated and real effort. In Figure 7 you see the component named Reversing_Lamp_Outage as highlighted. It has the values Entry=3, Exit=1, Read=3, Write=0, which amounts to CFP=7. Its estimated Code Size=894 Bytes and its Real Code Size=812 Bytes.

In Figure 7 you also see the categorization factor values for the component. Its categorization factor values are Team=A, Functionality=Comf & Conv, etc.

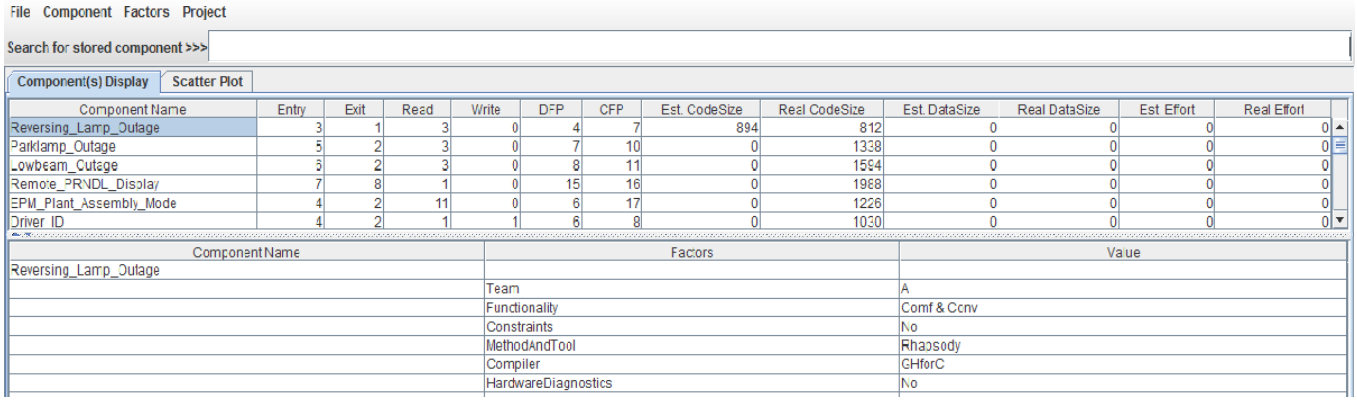

Figure 7. Components Display Tab

Estimations are based on components with the same set of factor values. From the set of components the CFP value and the Code Size is selected and a linear model is created using linear regression. In the “Scatter Plot” tab of the upper section of the window in Figure 7 these values and the linear model is presented when the user is estimating a component, see Figure 8. The scatter plot has CFP on the X axis, and Code Size in Bytes on the Y axis. The squares in the plot show historical data for component implementations, and the line shows the linear model calculated using linear regression on the historical data. To the right in the Scatter Plot Tab, the category name (Category=Series 2 in this case) of the historical data is shown. As many components as possible should be filled with historical data as more values will result in a better linear model. The absolute minimum is 2 valid components in a set for estimation. This would most likely result in estimation with an unsatisfying margin of error, so to guide the user the estimation results contains information about the R^2 value (R^2=0,99846) of the linear regression and the number of valid components used (#Data=10). With experience the engineer should be able to tell by these values if the margin of error of the estimation is within acceptable borders for the current task. The equation of the linear model is also shown (y=126*x+12,4). In the lower part of the Scatter Plot Tab we see that the Reversing_Lamp_Outage component has been estimated, that Category=Series 2 has been used for the estimation, that CFP=7, and that the estimated Code Size=894 Bytes. If the user is satisfied with the result it can be copied to the component by using the button in the bottom of the Scatter Plot Tab.

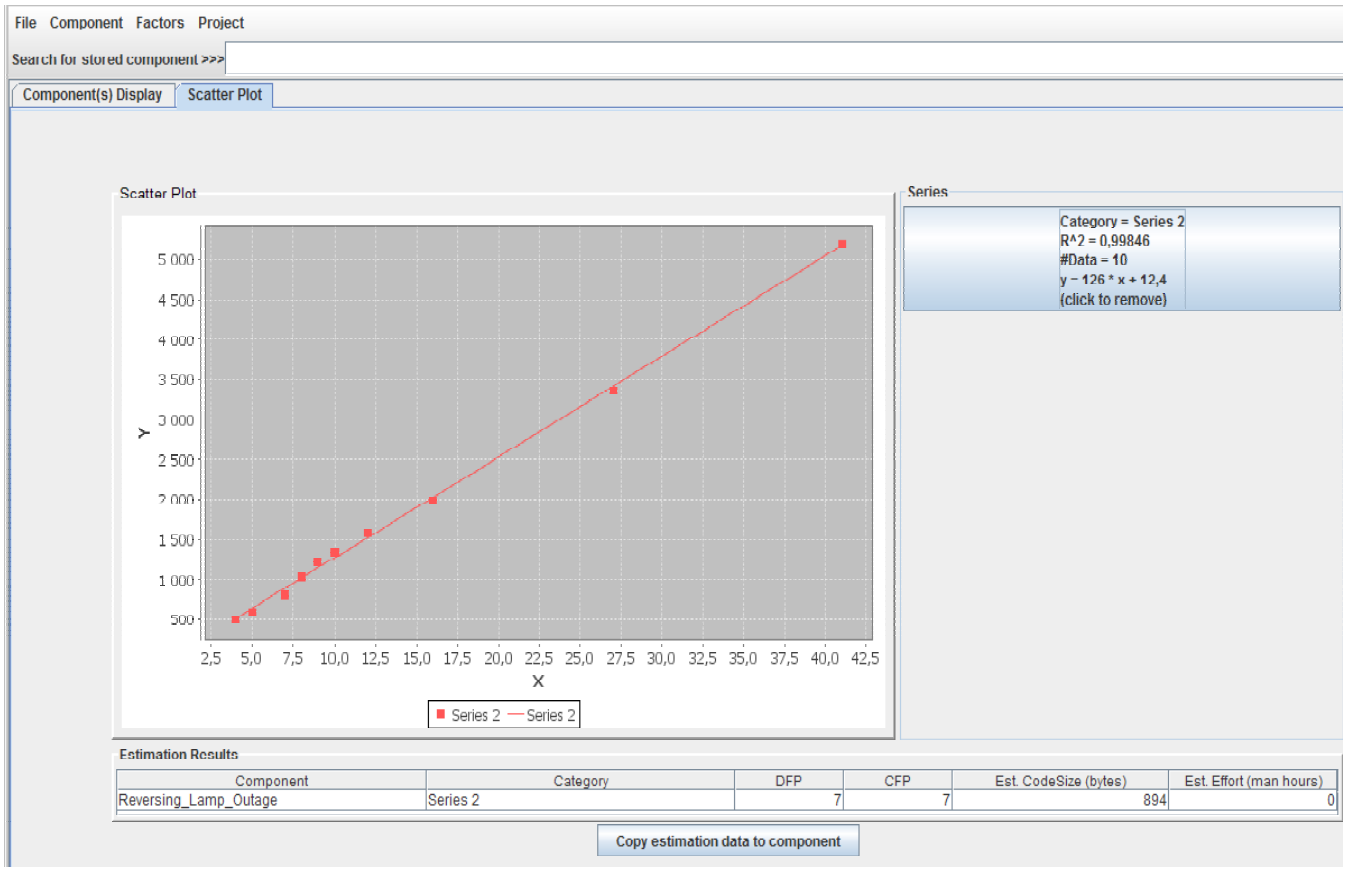

Figure 8. Scatter Plot Tab

The Components Display Tab and the Scatter Plot Tab are the main windows in the tool. Other windows in the tool are designed to guide the user how to perform a certain task. Examples are the Factors Manager Tab (where categorization factors can be added or removed), the Estimation Setup Tab (where estimations are initialized by selecting a component in the "Component Display" and then the Category and type of estimation (Code Size or Effort) can be selected), and the Add Category Tab (where categories can be defined as a list of factors and corresponding factor values).

B. CompSize tool architecture

The architecture of the tool was designed to be able to modify for updating or enhancing the functionality. It is structured in three layers to separate the main logic of the tool from the mapping rules and user interface.

The main logic layer contains data about all components with their information, and categories with their factor values. It contains code for the COSMIC calculations and the statistical calculations. The main logic layer should remain stable from changes in the user interface, mapping rules for certain types of input, etc.

The mapping rules are implemented in the IO layer. The mapping rules are tailored to extract information from input formats like Component Diagrams, Sequence Diagrams, etc. Our current version of the tool only have mapping rules for Component Diagrams, but the mapping rules are separated to allow for adding additional mapping rules in the future. The IO layer is also responsible for working with files such as extracting information from XMI files, or storing data like component names and values, categories with factor values, etc in XML files.

The user interface is designed to guide the user what to do next. Each window has a specific purpose, as described in the previous sub-section. Estimation results can be viewed in scatter plots or in table format.

The tool was validated during development to make sure that the different parts work as expected. Demo workshops were conducted to get feedback about the user interface from potential users of the tool. In the next section we will describe a case study we conducted to evaluate the tool.

V. CASE STUDY

This section describes a case study we conducted to evaluate the UML Profile and the CompSize tool. The case study was performed using requirement specifications and component diagrams together with the corresponding implementations from Saab. The purpose is to demonstrate that the UML Profile can capture the information needed for CFP measurement and Code Size estimation, and to verify that the CompSize tool is able to import the information modeled using the UML Profile. The calculation parts of the tool were validated during the development of the tool.

During the case study, we performed all the steps of our Code Size estimation process for 20 components of two categories. We illustrate the process by going through the Truck Bed Cargo Lamp example we have used earlier in this paper. For the sake of repetition; Figure 2 and Figure 3 show the input to our estimation process, Figure 4 shows how the input information is mapped onto the COSMIC Generic Software Model, Figure 5 shows how the input information is modeled using the UML Profile, and Figure 6 shows how we modeled the same information in Rhapsody.

Next step was to export an XMI file from Rhapsody containing the information modeled using the UML Profile. The CompSize tool imported the information, identified the data movements, calculated the CFP value to 7, and identified the categorization values. The result is shown in Figure 9.

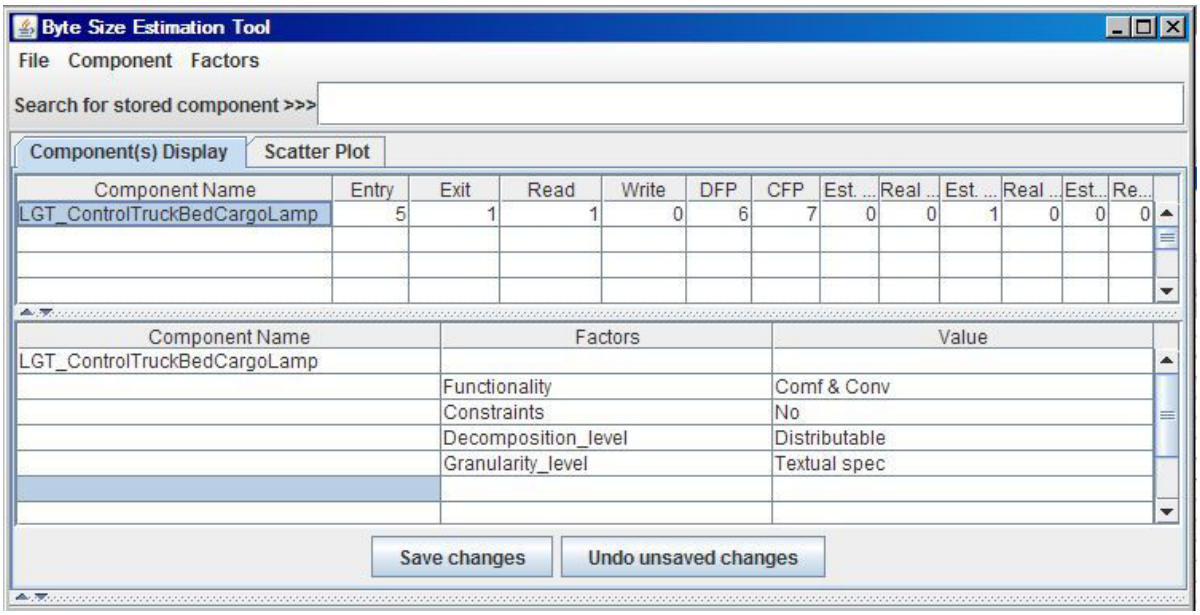

Figure 9. The information from the UML model imported into the CompSize tool.

In Figure 9 we see that the name of the component is imported correctly, as well as Entry=5, Exit=1, Read=1, and CFP=7. The categorization factor values (Functionality =Comf & Conv, Constraints=No, etc.) in Figure 9 are used to select the proper linear regression model to convert the CFP value into Bytes, and hence estimate the implemented Code Size of the distributable component. The same process was performed for 20 components available at Saab, and in a realistic setting. The results were similar as for the LGT_ControlTruckBedCargoLamp component. Hence, we conclude that the UML Profile can capture the information needed for COSMIC, and that the tool can import this information from an XMI file.

VI. Related Work

Marin et al. [30] presents a survey of existing literature related to measurement procedures based on COSMIC FP. Eleven procedures are presented of which two applies to the real-time systems domain. Of these two, the most relevant one [6] uses models developed in the ROOM (Real-time Object Oriented Modeling) language as input to the µcROSE tool [7]. The other procedure uses class, state, and collaboration diagrams specified using xUML, as input for manual measurement of COSMIC FP.

Lavazza and Del Bianco [20] shows how to use UML use case, component, class, and sequence diagrams to improve the practice of COSMIC measurement. Lavazza and Robiolo [21] propose how to measure functional complexity from UML sequence diagrams.

[22] and [36] report ongoing activities with the purpose of automating the FP measurement. In [22], it is described how UML sequence diagrams can capture the information needed as input to the COSMIC FP method. This facilitates the possibility to automate the calculation of FP, by extracting the number of messages exchanged in the sequence diagrams. In [36], it is shown that UML class diagrams and sequence diagrams can be used as input for a software tool that automatically calculates the IFPUG FP.

Fornaciari et al. [8] describes how IFPUG FP can be used to estimate the total development cost, and to optimize the partitioning of hardware and software based on cost and reusability. The starting point for the proposed method is a system-level description in UML class diagrams and sequence diagrams.

Stern [33] reports on lessons learned from using COSMIC FP for effort estimation purposes at Renault automotive company. Strong correlation (R^2=0,93) were found between CFP and supplier effort invoice data. Stern and Gencel [34] investigate the relationship between COSMIC FP and memory size of functions using data from the automotive industry. They found strong correlation (R^2=0,99) in a range of Functional Sizes from CFP=7 to CFP=748. This confirms our own results reported in [23],[24],[25],[26],[27],[28].

VII. Conclusion and Further Work

In this paper, we presented the CompSize tool that we have developed, and that is based on the mapping rules between CFP and the available information we have established in our previous work. We use our UML Profile capturing all information needed for CFP measurement and estimation of Code Size, and show how to model this information in the IBM Rational Rhapsody tool. From Rhapsody we show that the CompSize tool can import the modeled information to achieve automated estimation of Code Size based on CFP.

Besides the increased efficiency obtained by our automated estimation approach, we expect to increase repeatability and consistency in the estimation process compared to a manual approach. We conduct a case study using requirement specifications and software implementations from the automotive industry to evaluate the tool.

The tool is structured to allow having different parsers adapted to different types of input, like use cases and sequence diagrams, etc. We plan to investigate the mapping between COSMIC and other types of input types as future work, and to implement parsers for those input types.

We will also apply the tool in the ordinary development process at Saab, to collect more historical data and to improve the estimation accuracy. A natural next step is also to apply the tool in other domains where embedded software is developed. This would further evaluate our UML Profile and the CompSize tool.

References

[1] Albrecht, A.: Measuring application development productivity. Proc. of the IBM Applications Development Symposium, Monterey, CA, 83-92, Oct. 1979.

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

[3] Albrecht, A.: AD/M Productivity Measurement and Estimate Validation., IBM Corporate Information Systems, IBM Corp., Purchase, NY, 1984.

[4] Baillargeon, R, and Flores, R.: From Algorithms to Software – A Practical Approach to Model-Driven Design., SAE paper 2007-01-1622.

[5] COSMIC (The Common Software Measurement International Consortium), The COSMIC Functional Size Measurement Method, Version 3.0.1, Measurement Manual, 2009.

[6] Diab, H., Frappier, M., and St-Denis, R.: Formalizing COSMIC-FFP Using ROOM. ACS/IEEE Int'l conf. on Computer Systems and Applications, (2001).

[7] Diab, H., Koukane, F., Frappier, M., and St-Denis, R.: µcROSE: automated measurement of COSMIC-FFP for Rational Rose RealTime. Information and Software Technology 47(3), pp. 151-166, (2005).

[8] Fornaciari, W., Micheli, P., Salice, F., and Zampella, L.: A First Step Towards Hw/Sw Partitioning of UML Specifications. Proc. of the Design, Automation and Test in Europe Conf. and Exhibition (DATE'03), pp. 668-673, (2003).

[9] Gencel, C, and Demirors, O, "Functional Size Measurement Revisited", ACM Trans. Softw. Eng. Methodol. 17, 3, Article 15 (June 2008).

[10] Gencel, C., Heldal, R., and Lind, K.: On the Relationship between Different Size Measures in the Software Life Cycle. Proc. of the IEEE Asia-Pacific Software Engineering Conference (APSEC 2009), pp. 19-26, (2009).

[11] IBM Rational Rhapsody, http://www.ibm.com/.

[12] IFPUG, Function Point Counting Practices Manual, Release 4.1, IFPUG, Westerville, OH, 1999.

[13] ISO/IEC 14143-1:2007, "Information Technology - Software Measurement - Functional Size Measurement - Part 1: Definitions of concepts", 2007.

[14] ISO/IEC 14143-2:2002, "Information Technology - Software Measurement – Functional Size Measurement - Part 2: Conformity Evaluation of Software Size Measurement Methods to ISO/IEC 14143-1", 2002.

[15] ISO/IEC TR 14143-3:2003, "Information Technology - Software Measurement – Functional Size Measurement - Part 3: Verification of Functional Size Measurement Methods", 2003.

[16] ISO/IEC TR 14143-4:2002, "Information Technology - Software Measurement – Functional Size Measurement - Part 4: Reference Model", 2002.

[17] ISO/IEC TR 14143-5:2004, "Information Technology - Software Measurement – Functional Size Measurement - Part 5: Determination of Functional Domains for Use with Functional Size Measurement", 2004.

[18] ISO/IEC 14143-6:2006, "Guide for the Use of ISO/IEC 14143 and Related International Standards", 2006.

[19] ISO/IEC 19761:2003, "Software engineering - COSMIC-FFP - A functional size measurement method", 2003.

[20] Lavazza, L., and Del Bianco, V.: A Case Study in COSMIC Functional Size Measurement: The Rice Cooker Revisited. Int'l Workshop on Software Measurement (IWSM), November 4-6, (2009).

[21] Lavazza, L., and Robiolo, G. Introducing the Evaluation of Complexity in Functional Size Measurement: a UML-based Approach. Int'l Symposium on Empirical Software Engineering and Measurement (ESEM 2010), (2010).

[22] Levesque, G., Bevo, V., and Tran Cao, D.: Estimating Software Size with UML Models. Canadian Conference on Computer Science & Software Engineering (C3S2E '08), pp. 81-87, (2008).

[23] Lind, K., and Heldal, R.: Estimation of Real-Time System Software Size using Function Points. Proc. of the Nordic Workshop on Model Driven En-gineering (NW-MoDE), pp. 15-28, (2008).

[24] Lind, K., and Heldal, R.: Estimation of Real-Time Software Code Size using COSMIC FSM. Proc. of the IEEE Intl. Symposium on Ob-ject/component/service-oriented Real-time distributed Computing (ISORC 2009), pp. 244-248, (2009).

[25] Lind, K., and Heldal, R. Estimation of Real-Time Software Component Size. Nordic Journal of Computing (NJC), no.14, pp. 282-300, (2008).

[26] Lind, K., and Heldal, R.: On the Relationship between Functional Size and Software Code Size. Proc. of the Workshop on Emerging Trends in Software Metrics (WETSoM´10) held in conjunction with the Intl. Conf. of Software Engineering (ICSE'10), (2010).

[27] Lind, K., and Heldal, R.: Categorization of Real-Time Software Components for Code Size Estimation. Int'l Symposium on Empirical Software Engineering and Measurement (ESEM 2010), (2010).

[28] Lind, K., and Heldal, R.: A Practical Approach to Size Estimation of Embedded Software Components. Approved for publication in IEEE Trans. Softw. Eng. (2011).

[29] Lind, K., and Heldal, R.: A Model-Based and Automated Approach to Size Estimation of Embedded Software Components. Accepted to be presented at ACM/IEEE The 14th Intl. Conf. on Model Driven Engineering Languages and Systems (2011).

[30] Marin, B., Giachetti, G., and Pastor, O.: Measurement of Functional Size in Conceptual Models: A Survey of Measurement Procedures Based on COSMIC. Int'l Workshop on Software Measurement (IWSM), November 18-19, (2008).

[31] OMG, Unified Modeling Language (UML), Superstructure Specification, V2.3, Object Management Group, http://www.uml.org/.

[32] OMG, Unified Modeling Language (UML), Infrastructure Specification, V2.3, Object Management Group, http://www.uml.org/.

[33] Stern, S.: Practical experimentations with the COSMIC method in Automotive embedded software field. Int'l Workshop on Software Measurement (IWSM), November 4-6, (2009).

[34] Stern, S., and Gencel, C.: Embedded Software Memory Size Estimation Using COSMIC: A Case Study. Int'l Workshop on Software Measurement (IWSM), November 10-12, (2010).

[35] Szyperski, C, "Component Software: Beyond Object-Oriented Programming." 2nd ed. Addison-Wesley Professional, Boston 2002 ISBN 0-201-74572-0.

[36] Uemura, T., Kusumoto, S., and Inoue, K.: Function-point analysis using design specifications based on the Unified Modelling Language. Journal of Software Maintenance and Evolution: Research and Practice 13, 4, pp. 223-243, (2001).