Integrating System Modeling and Cost Models Using Meta-Modeling Techniques

Viktor Steiner

Evopro Innovation Ltd.
Budapest, Hungary
steiner.viktor@evopro.hu

Gergely Mezei

BUTE DAAI
Budapest, Hungary
gmezei@aut.bme.hu

Abstract. The precise estimation of time and resource consumption plays a pivotal role in planning software development projects at their earliest development phase. Since cost parameters are mostly determined by the architecture, a possible approach is to design a platform independent architectural model of the prospective software and estimate the cost based on it.

In this paper, we introduce a method, which produces a cost estimate by processing the architectural model of the software being designed. The provided method analyzes the architectural models, and utilizes a modified version of Function Point Analysis to determine the probable cost based on the analysis. The paper also presents a preliminary verification process to evaluate the accuracy of the cost estimation method. The main achievement of the introduced method is that it estimates cost in platform independent units, which can be refined to give accurate cost estimation for different platform implementations.

Keywords: Meta-modeling, Cost Estimation, Function Point Analysis

1 Introduction

Cost estimation is a major challenge in software industry. It is hard to find features that can precisely predict the expected cost of the complete development process. Since architecture has the greatest impact on development costs, architectural models can provide a basis for the estimation. In this paper, we provide a method, which analyzes the architectural models created in the design phase and estimates the expected cost of the software to build.

Our solution is unique among cost modeling methods, since it applies Multi-Paradigm Modeling techniques to achieve its goal. Compared to other existing cost estimation techniques, our approach does not require creating a separate cost model manually. Instead, we analyze architectural models and generate the cost model from them. Our method maps model elements of the software architecture domain to the concepts of the cost modeling domain. Since we use a platform independent architectural and cost modeling domain, our results can be applied early in the development process, before deciding which technology to use for the implementation. This also means that our method is also useful for facilitating the decision between possible implementation

technologies, because the cost estimation result can be refined into estimations on different platforms and compare the cost predictions.

The paper is organized as follows: In Section 2, we give a short summary of the state of the art in the field of cost estimation approaches. Section 3 introduces the Visual Modeling and Transformation System [1], which we used to implement the cost estimation method. In Sections 4 and 5, our cost estimation method is presented. The method is a modified version of Function Point Analysis [2], adapted to SysML [3] models. Section 6 introduces a verification process, which is used to test the accuracy of the results provided by the cost estimation method. Finally, Section 7 concludes the outcome of our work and gives main directions of future work.

2 Related Work

Existing cost estimation methods typically do not use existing resources such as requirements, specification, or architectural models for calculating the probable cost, they use their own cost model, which must be prepared separately. This is problematic for various reasons, e.g. (i) It takes extra time and effort to estimate probable development cost. (ii) Cost estimation becomes a mostly manual task, since the cost model does not rely explicitly on existing resources. Manual steps increase the probability of errors in the estimation. (iii) A cost modeling expert is always needed, who prepares and analyzes the cost model. These issues arise in most of the existing cost estimation methods, for example, in COCOMO II [4].

However, there are some methods, which use resources from the development process. An estimation technique that measures development effort based on use cases [5] and another that uses requirements as a basis [5] can be mentioned here. Although these methods use available development resources, they still need too many manual steps to produce the estimation. This is because both requirements and use cases are high level concepts, which can hardly be formalized, which could enable programmatic analysis. On the other hand, use cases and requirements are available very early in the development process, therefore the estimation can be performed earlier compared to our method. However, performing the analysis on architectural models can be much more accurate, since more formalized data is available. Since these two approaches can both be performed during the development process, they can complement each other, by giving a vague initial estimation, and then later calculate a refined, more accurate estimation.

Although we had not found methods that estimate development cost from architectural models, there are some methods that are based on similar concepts, of which [6] is the most closely related. The method uses architectural models to predict performance, and to facilitate architectural design decisions. The latter is among our goals as well, as our method can be used to compare and evaluate different architectural versions based on their estimated costs.

3 The modeling environment

A cost estimating algorithm requires a modeling environment, in which architectural system models can be created, models can be processed and analyzed programmatically. We had chosen SysML [3] to model the architecture. The main reason behind this decision was that SysML is a widely used and accepted general purpose systems modeling language, and we used it in several projects previously in Evopro Innovation Ltd. However, our method is only partially specific to SysML, it can be applied with other architecture modeling languages as well.
The selected modeling tool, in which the SysML language environment was created, is the Visual Modeling and Transformation System [1]. In VMTS, any modeling language can be defined by creating its metamodel. The framework offers a highly customizable workbench to edit the models visually and the models can be processed programmatically using the VMTS Domain Specific Language API.
Our SysML dialect was defined by a meta-model based on the OMG SysML and the related parts of the UML specification. Creating the whole meta-model of the UML and SysML languages was not our goal, we intended to calculate our cost estimation in the architectural design, thus, we focused on those parts of the UML/SysML meta-models that describe the architecture. As described later, in sections 4 and 5, we identified the following aspects of SysML as required: (i) the Block Definition Diagram, (ii) the Internal Block Diagram, (iii) the Requirement Diagram, (iv) the Use Case Diagram and (v) the Sequence Diagram. As the first step of our work, we created these languages and customized their visual appearance and behavior according to the SysML standard.

4 Cost estimation

In the past decades, different methods were developed for software cost estimation. Our goal was to find the best suitable method among these for our purposes. The selected method had to be: (i) current, (ii) used in software industry (to ensure that it predicts the development cost correctly) and (iii) publicly documented to avoid copyright issues. Moreover, we have decided to focus on solutions capable of estimating the size of systems created with object oriented principles. Finally, we have chosen the method described in [7]. [7] describes a collection of methods, each usable in different phases of software development projects. We only needed the ones that deal with calculating the size of the software, since the size has the greatest impact on development costs, and it can be measured in the architectural design phase. In our method, the cost is estimated based on the software size, which is typically measured in two ways: (i) Source Line of Code (SLOC) and (ii) Function Points. In case of SLOC, the number of source lines required to implement the software in a particular language is measured. In contrast, function point measuring methods quantify the functionality of software in an abstract, platform independent unit. We selected the later one, since it is platform and technology independent. Moreover, function points, despite they are abstract measurement units, can be converted to an estimated number of source lines, based on past development experiences.

4.1 Function Point Analysis

Function point measuring methods are collectively referred to as Function Point Analysis (FPA) methods. FPA has no official standard, several different implementations exist. In our solution, we used the approach described in [7], and a more detailed version of the same method in [2]. As we mentioned before, FPA measures the size of software based on its functionality. In FPA's interpretation, functions of a software are always transactions, which are executed on some kind of data set. Therefore, function point count is determined by logically related data sets, and by the transactions associated to them. The basic terms of FPA can be seen on Fig. 1.

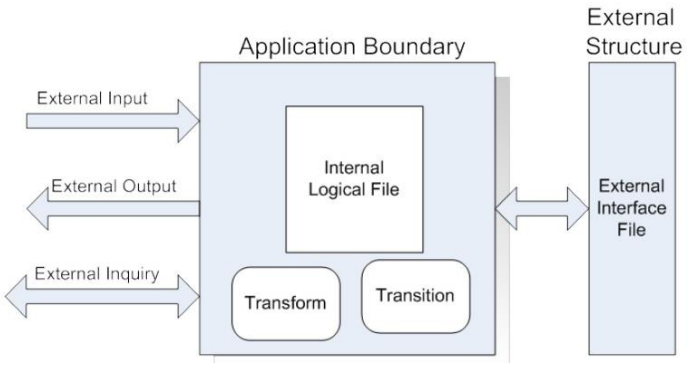

Fig. 1. Overview of the basic terms of Function Point Analysis

We modified the original Function Point Analysis to adapt it to SysML architectural models. Firstly, we examined, which of the basic FPA terms are necessary in order to implement the method properly:

- ***Application Boundary:*** It specifies the communication interface between the system and the outside world.
- ***Internal Logical File (ILF):*** Logically related set of data, maintained by the application.
- ***Transaction:*** An elementary process, which obtains data through the application boundary. There are three different kinds of transactions we distinguish:
  - ***External Input (EI):*** A transaction obtaining data from the environment
  - ***External Output (EO):*** A transaction submitting data to the outside environment.
  - ***External Inquiry (EQ):*** A transaction, which gets data from inside of the application, through the application boundary. The data is queried according to query parameters. The requested data cannot be derived (calculated) data.

We discovered that the above terms are necessary to implement the method, and can be matched to SysML concepts, as described in section 4. However, the remaining terms from Fig. 1. are not needed for the implementation. External Interface File (EIF) is an ILF, maintained by another application. It is not part of our model, because it is not an Object Oriented Programming concept and our focus was on estimating the cost based on OOP software models. Transformation & Transition: A transformation is a sequence

of mathematical calculations transforming the input data into the required form. A transition is an event, which changes the state of the application. These concepts can clarify the results of the estimation, but they are not elaborated in the architectural design phase.

5 Mapping Function Point Analysis to SysML

In this section, we present how we managed to map the basic FPA terms into SysML concepts, and calculate function points by analyzing SysML models.

5.1 Application boundary

The first step of FPA is to define the application boundary. Here we have to analyze the data used by the application, and determine whether it is maintained by the application. If it is, then the data resides inside the application boundary, otherwise it belongs to the outside environment. When we design the architecture of software in SysML, the first step is the definition of the application boundary, however, it is not displayed explicitly as a model item: The first step of software modeling is usually the definition of functionality, in the form of Use Case diagrams. In Use Case diagrams, actors belong to the environment, and the highest level use cases, which they are associated with, are matched to FPA transactions that obtain data through the application boundary.

5.2 Data types

The second step of FPA is to identify and rate the data sets maintained by the application. These data sets always appear as an ILF. An ILF is a user identifiable group of logically related data that resides entirely within the application boundary, and is maintained by External Inputs. An ILF has an inherent meaning, it is internally maintained, it has some logical structure and it is stored in a file, as defined in section 9 of [2]. According to this definition, ILFs are almost identical to persistent entities of an OOP application, whose structure and connections can be modeled on an Entity Relationship diagram, or in our case, on a SysML Block Definition diagram. However, this data model must be programmatically distinguishable from the other system elements that are also modeled on Block Definition diagrams. This can be achieved by performing a small modification on the original SysML meta-model and adding an attribute – a flag – to the Package meta element. FPA analyzer can decide whether to search for the data model elements in those Packages, or not.

After identifying ILFs, the next step is to evaluate their complexity and rate them. Complexity analysis is based on two concepts: (i) A Record Element Type (RET) is a user recognizable sub group of data elements within an ILF. (ii) A Data Element Type (DET) is a unique, user recognizable, non-recursive (non-repetitive) and dynamic field in a RET. Additionally, a DET can invoke transactions or can act as additional information regarding transactions. During the evaluation process, Record Element Types and Data Element Types within an ILF are counted.

By definition, an ILF itself is also a user recognizable group of data elements, therefore it always consists of at least one RET. ILFs consist of more than one RET, if they contain multiple logically related sub groups of data, which have no meaning on their own and can only be interpreted inside the ILF. The RET concept can be illustrated using the following two cases:

- There are two logically related sets of data (A and B), which have a common subset, a key field, by which they are connected to each other. Both A and B can be interpreted on their own, thus they are both considered a separate ILF.
- There are two sets of data (A and B), where B is a subset of A. In this case, B cannot be interpreted on its own. Therefore, it cannot be an ILF, it can only be considered a RET inside A. An example for this case is a music CD that contains songs. Both the CD and the songs have attributes, but the songs cannot be interpreted on their own, without the data contained by the CD.

When interpreting the RET concept on SysML models, we assumed that the data model of the designed application is available in the form of SysML Block Definition diagrams. The persistent entities of the data model can be interpreted as ILFs or RETs, as it was mentioned above. According to the definition, a data set can only be considered a RET if it is a real subset of an ILF. This means that an entity in the data model can only be considered a RET if it has only one parent, and its children are RETs. If the above condition is satisfied, an entity is considered a RET, if not, it is considered an ILF. Note the parent-child relation here means the usual one-to-many relation used in Entity-Relationship diagrams.

The Data Element Type concept can be easily mapped to SysML models. According to the definition, a DET is very similar to a data field of a persistent entity. Since we have already identified ILFs and RETs, the only remaining task is to count the data fields for each identified RET, and the evaluation of the data model is complete. The actual weight of an ILF can be read out from the corresponding cell of the table defined in Section 9 of [2].

5.3 Transactions

The third step of Function Point Analysis consists of the identification and evaluation of transactions. Our method is based on Use Case diagrams of the system at this step. As it was pointed out previously, top level use cases – which are directly connected to actors – represent transactions, thus their automatic identification is easy. On the other hand, programmatic determination of the transaction kind (External Input, External Output or External Inquiry) is not possible. This is mainly because if we want to distinguish between the types, we need to

- **Determine the main direction of the data flow.** This would distinguish input transactions (EI) from output transactions (EO and EQ). Here we use the term “main direction” instead of simply using “direction” because according to FPA definition, all three transaction types can send data in both directions. For example, an input transaction can send back a status code, which indicates whether the transaction is

successful or not. In SysML, however, we can only show the direction(s) in which data flows, there is no such concept as main direction.

- **Check whether the output data is derived or not.** This would distinguish between output transaction types (EO and EQ). Based on FPA definition, derived data is the result of some kind of calculation. In case of SysML models, the only indicator of data derivation is that data type changes during the execution of a transaction. This, however, is not an accurate conclusion, because it is possible that no calculation is performed, the data is just transformed into another form. For example, an array of items is transformed into a linked list of the same items. In this case, type of the data is changed, but the actual information remains the same.

Consequently, in case of SysML models, we cannot distinguish FPA transaction types from each other. Because of this, we decided to give up the idea of fully automated cost analysis, and add some additional information to the SysML meta-model, which helps identifying transaction types. As mentioned before transactions are mapped to top level Use Cases in SysML models. Therefore, we decided that the most optimal solution is to add an attribute to the Use Case meta-element, which marks the transaction type. After setting this attribute, the evaluation of transactions can be performed automatically. The process consists of two steps: counting of (i) data element types and (ii) referenced file types.

The first step is to calculate the data element types. Parameter and return values get into and out of the application through the application boundary. In case of OO applications, the boundary is usually an interface, whose operations start the execution of transactions. In SysML, this concept can be modeled as Use Case and Sequence Diagrams, where there is a Sequence Diagram associated to each use case shown on the Use Case Diagrams. The Sequence Diagrams show the order of operations that implement the particular use case. Here we only analyze the first operation of a sequence, which is the interaction point between the application and the environment. The parameter and return types of that operation are used to calculate the complexity of the transaction being analyzed, therefore, these are the types that have to be counted. Note that there are transactions that cannot be analyzed this way. For example, take a transaction, which queries the database for some data, and displays the results on the GUI. The operation that triggers the transaction does not give back a return value, it just updates some part of the GUI, but it is clear that data gets through the application boundary. We solved this problem by creating a new descendant of the Operation element in the SysML meta-model, called GUIOperation. On this kind of operation, the modeler can set the properties that it updates, thus, the complexity of the function can be fine-tuned.

The second step is to count the file type references that are (by definition) unique ILFs that a transaction references during its execution. To count them, the prerequisites are the same as they were in the previous step. Namely, each top level use case should have a corresponding Sequence Diagram, which shows the order of operations implementing the particular use case. When the necessary diagrams are ready, processing them to find referenced file types is an easy task. We only need to analyze the operations of a transaction, and count their parameter and return types. Each type is counted only once, because file type references are unique by definition.

By now we identified the transactions, and determined their types. We have counted the referenced file types and data element types, thus the evaluation process of transactions is complete. The actual weights of the transactions can be read out from the corresponding cells of the tables in Sections 5, 6 and 7 of [2].

5.4 Evaluation

The last step of FPA is to calculate the Function Point count, by using the following formula [2]:

$$FP = \sum_{i=1}^{m} ILF_i + \sum_{i=1}^{n} EI_i + \sum_{i=1}^{o} EO_i + \sum_{i=1}^{p} EQ_i, \quad (1)$$

where ILF_i, EI_i, EO_i and EQ_i are the weights of the files and transactions that were calculated according to the methods defined in earlier sections. The resulting value is a platform independent quantity that not only measures software functionality, but it can be converted to platform specific source code estimations as well. For the conversion, a table is used, which is maintained and updated by Quantitative Software Management Inc. [8] The table contains factors to convert Function Points into SLOC estimation on different programming languages. The conversion factors are based on historical data from completed software projects (currently 2192 different projects). The converted SLOC values provide a basis for comparing source code estimations on different platforms, and selecting the most suitable platform.

Note that the original FPA method uses a Value Adjustment Factor to fine tune the Function Point count, based on non-functional requirements. This adjustment can be done with our result as well, as described in Sections 11 and 13 of [2].

6 Verification

In order to check that our method produces correct results, a verification method was implemented. Our original plan was to apply the method from the beginning of a new project and validate its results after completing the project. We realized that this would require months to apply and we had difficulties in convincing the project management. We had strict time constraints in the current projects and the project management also wanted to have a preliminary validation of the results before introducing the proposed method in real development. We have decided to use a simpler but not as precise method: we generated SysML models from the source code of projects already completed. We used the source code to generate an architectural model from a complete software and run the cost estimation method on it, thereby verifying its accuracy. We were aware that the accuracy in this case depends on how precisely the generated model complies with the source code, and how much the generated models differ from the architectural models created in the design phase. We made assumptions (e.g. the architecture does not change in a large extent during the development) keeping in mind that the verification method is only preliminary and it is necessary to prove the correctness of our method, which can be fine-tuned later, once it is used in production scenarios.

As the subject project, we used an mCPS system developed by Evopro Innovation Ltd. [9] The system consists of the following components: (i) database, (ii) server, (iii) a thick administrator client and a (iv) mobile client. The first three components were using Microsoft technologies (Microsoft Azure, .NET WCF and WPF), while mobile client applications were implemented on every significant mobile platform (Android, iOS, Windows Phone, Windows 8). We parsed the source code mainly by the open source tool, NRefactory [10] which produced an AST. From the AST, we generated the architectural model by using the DSL API of VMTS. Note that the method used here is not specific to this case study, it can be applied on any projects written mainly in C#.

We created two test cases, i.e. a combination of components that build up the test system, on which the verification process is performed. We have defined the following two test cases: (i) Database + server: here, the application boundary is an API, since the database and the server does not have any graphical user interface. (ii) Database + server + admin client: in this case, the application boundary is the user interface of the admin client. After performing the verification process, we compared the real and the estimated source lines of code (SLOC). We calculated estimated SLOC values from function points based on the previously mentioned table [8]. However, there is no way to convert a fraction of a FP value to an SLOC estimation in language 1 and the remainder to language 2. This affects the accuracy of the estimation, because the tested application is not a pure C# application in either test case, there are some T-SQL and XAML language parts in the components as well. The estimated SLOC values, however, are probabilistic values (most likely, minimum and maximum values), which compensates the above inaccuracy of the conversion factors.

In test case 1 (Database + server), we identified 421 Function Points, which is translated to **12209** – **29470** (most likely **22734**) lines of C# code. The real application consisted 19696 lines of C# and 3055 lines of T-SQL code (**22751** in all). In test case 2 (Database + server + admin client), the result of the estimation was 699 Function Points, which is converted to **20271** – **48930** (most likely **37746**) lines of C# code. The actual SLOC values were 38365 lines of C#, 3055 lines of T-SQL and 5056 lines of XAML code (**46476** in all).

As it is shown above, the first estimation is almost exactly the same as the most likely estimation value. The difference is less than 0,1%! The second estimation is not that accurate, but the estimation is between the limits. After analyzing the verification results, we discovered that the cause of the relative inaccuracy in the second test case was the amount of XAML code, since a high percentage of the code was duplicated. This indicates that the verification algorithm should be enhanced with the capability of detecting code duplication. Apart from this, the results were convincing, the project management decided that the method is ready to be tested in production environment.

7 Conclusion

We designed and implemented a method that analyzes architectural models, and provides a cost estimation based on it. We did so because we discovered that nowadays, there is a great need for a cost estimation method that produces results automatically

based on already available resources. Our technique is implemented using VMTS [1], a meta-modeling tool capable of creating and processing architectural models in the SysML language [3]. The advantage of the solution is that it does not need separate cost models, information is extracted from the architectural models automatically. Note that although we rely on an extended version of SysML, the extensions are used to create a more precise architecture model and they are not only used by cost estimation.

By automatizing the cost estimation, there is no need for extra time and effort allocated to the cost estimation. Another benefit is that the method uses a modified version of Function Point Analysis [2], which produces a platform independent result. In this way, the estimation process can be performed before even deciding, on which platform the software should be implemented. The result of the estimation can also be refined into source code estimations on different platforms. In this way, possible implementations using different technologies can be compared and the best can be selected. Besides presenting the method, we elaborated a basic, preliminary verification method and discussed the results. Although the verification was not based on real production process, its promising results show that the method is worth to examine further. In the future, we plan to test the method on several production scenarios and use it in real development environments and add support for estimating the cost of mixed platform projects.

8 Acknowledgement

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred.

9 References

1. Visual Modeling and Transformation System (VMTS): https://www.aut.bme.hu/en/Pages/Research/VMTS/Introduction
2. David Longstreet: Function Point Training and Analysis Manual: http://www.softwaremetrics.com/Function%20Point%20Training%20Booklet%20New.pdf
3. OMG SysML 1.3: http://www.omg.org/spec/SysML/1.3/
4. Constructive Cost Model II (COCOMO II): http://csse.usc.edu/csse/research/COCOMOII/cocomo_main.html
5. Arlene F. Minkiewicz, Estimating Software from Use Cases & Estimating Software from Requirements: http://legacy.pricesystems.com/research/white_papers.asp
6. Steffen Becker, Heiko Koziolek, Ralf Reussner, The Palladio component model for model-driven performance prediction, Journal of Systems and Software, v.82 p.3-22, January, 2009
7. STSC, Software Development Cost Estimating Guidebook: http://www.stsc.hill.af.mil/consulting/sw_estimation/softwareguidebook2010.pdf
8. Quantitative Software Management Inc., Function Point Languages Table: http://www.qsm.com/resources/function-point-languages-table
9. mCPS - End to End Mobile Publication: http://www.evoprogroup.com/page/mcps
10. Daniel Grunwald, Using NRefactory for Analyzing C# Code: http://www.codeproject.com/Articles/408663/Using-NRefactory-for-analyzing-Csharp-code