Traceable Complexity Metric from Requirements to code

Gabriela Arévalo
CONICET
Avda. Rivadavia 1917
(1033), Buenos Aires, Argentina
garevalo@austral.edu.ar

Gabriela Robiolo
Universidad Austral
Avda. Juan de Garay 125
(1054), Buenos Aires, Argentina
grobiolo@austral.edu.ar

Miguel Martinez Soler
Universidad Austral
Avda. Juan de Garay 125
(1054), Buenos Aires, Argentina
miguelmsoler@gmail.com

ABSTRACT
Complexity metrics are one of the useful measurements to determine the quality of a target software. However, the measurements are only applied and isolated in one specific development phase and there are no metrics that can show the quality (and the evolution) of the same software artifact in all development phases. In this paper, we propose a methodology to use a traceable metric to measure software artifacts from requirements to source code. Specifically, we validate our approach on transactions defined in use cases and implemented in source code, and some initial results show that our approach is promising to solve the problem of tracing software artifacts during the software development process.

Categories and Subject Descriptors
D.2.8 [**Metrics**]: Complexity measures, Process Metrics.

General Terms
Measurement, Design, Verification.

Keywords
Traceability, Metrics, Transactions, Requirements, Source code

1. INTRODUCTION
UML models provide a useful means to have a controlled (clean) process from requirements to implementation levels for developers during software development. The most known models are Use-Case Model, Analysis Model, Design Model and Implementation Model [1]. These models are not independent one from each other, but they keep different relationships between them. For example, the Use Case Model describes the proposed functionality of a target system. The Analysis Model describes the structure of the system or application, describing the logical implemenation of the functional requirements identified in the Use Case Model. The Design model builds on the Analysis Model by describing in detail the structure of the system and how the system will be implemented. Finally, the Implementation Model represents the physical composition of the implementation in terms of subsytems and implementation elements (directories and files, including source code, data and executable files). Even when we have described briefly all the models, we observe that there are (implicit) dependences that tightly link one model to another one. These dependencies are named *traces* [2]. They can be defined through the historical or process relationships between elements that represent the same concept in the different models. We consider them as implicit because there are no rules to derive the dependences from one model to another one.

However, the possibility of tracing different elements and relationships between the models is important in terms of understandability and change propagation. It will mainly help developers to forth and back track any error and improvements in any development level in a system. But the process of tracing is not trivial because as we said the dependencies of the different elements are implicit.

During software development, we apply different metrics to the software elements and relationships to measure the quality of the software. Specifically in case of size or complexity metrics, which provides a measurement to a specific internal product feature in the target software, we have specific metrics for each specific model or phase of a system lifecycle. Briefly, Function Points (FP) [3] are applied to models during modeling/requirements phase, Use case Points [4], Transactions and Paths [5] are based on Use case Models, Chidamber-Kemerer's [6] metrics are design-oriented, ciclomatic complexity [7] is graph-oriented and Lines of code is code-oriented. Even when the choice of metrics models is wide, there are no metrics that could be applied and traced through the complete process of software development (from requirements to code).

This feature is a drawback to track the evolution of the software elements and relationships between them. The developers can define and measure the requirements during the first stages of an application development. They can also calculate requirements size and complexity [5], but these measurements will be only kept as information of this development phase, because it implies a particular feature (in our case, requirements) in a specific model. However, project leaders must have a global view of the system, and must be able to identify each software element (and its features) from requirements to implementation phases. It is important for them to know they were implemented (or not), decomposed in other software components, integrated in an existing one, its size and complexity and so on.

To cope with this requirement, we develop an approach based on two metrics proposed in Robiolo et al. work [5, 8]: Number of Transactions (*nT*) and Number of Paths (*nP*). They are based on use cases and computed from textual descriptions of use cases and UML Models. *nT* is the number of transactions in the use case identified from the actor stimulus in the text by the verbs, i.e. the actor actions interacting with the system. *nP* is the number of paths of a transaction of the use case textual description.

Summarizing, *nT* measure the use case size and *nP,* the transaction complexity

In this paper, we show how both metrics can be calculated in source code, and we show how the traceability of the metrics helps to understand how the different requirements were implemented in a target application, and which are the explicit differences between requirements in terms of models and source code.

This paper is structured as follows: Section 2 details our approach to analyze the source code to obtain the metrics in the system. Section 3 presents the case study, the obtained measurements and analyze the results. Section 4 cites some related work, and finally Section 5 presents some conclusions and future work.

2. OUR APPROACH

As mentioned previously, we need to calculate two metrics: Number of Transactions (nT) and Number of Paths (nP) starting from source code using the concept of complexity metrics.

The starting point is the identification of a transaction in source code. To map the definition of a transaction based on use cases into code, the key aspect was to identify the actor' stimulus in code, named as *access-point*. As a transaction is made up of a set of sequential method calls, the access-point is the method that is not called but it calls other method. Thus, the complexity of a transaction would be computed in terms of number of the paths. Each transaction has a principal path, built with the chain of method calls from the starting point to the end of the transaction, and alternatives paths, which are identified by the "*if-then*" expressions. This definition of transaction is similar to the one suggested in [9], which makes cyclomatic complexity easy to compute, where each alternative path adds one to the complexity of the principal path whose complexity is also 1. In terms of formulas, if a transaction T has a chain of methods calls $m_1,\ldots m_k$, then

$$nP(T) = 1 + \sum_1^k CC(m_i) - k$$

where k is the number of methods calls, CC(mi) is the complexity metric of the method m_i.

The identification of transactions in source code is not a trivial work. Therefore, we developed an approach consisting of 5 steps:

1) Mapping from Source Code to a Metamodel. Our goal is not to link our approach to a specific programming language. Thus, instead of analyzing the source code itself, we generate a model of the source code to keep our analysis independent of the target source code. Our case study is built with Java, so we chose *Recoder* to generate the model of the source code. Recoder is a Java framework for source code metaprogramming aimed to deliver a sophisticated infrastructure for many kinds of Java analysis and transformation tools [10].

2) Computation of Mc Cabe Complexity of Methods. As we base our computation of the nP on the Mc Cabe Complexity, we calculate it for each method using the Abstract Syntax Tree (AST) generated by Recoder.

3) Mapping AST to logical facts. Even when Recoder tool provides complete information regarding the target source code, we just need the information regarding method calls (to detect the access points to build the transactions) of the generated Abstract Syntax Tree (AST). Thus, we only keep the information of classes, methods, attributes, method calls (including constructors) and inheritance relationships, and the corresponding Mc Cabe complexity of all the methods. The metric is relevant in the computation of nP.

In a further step, we need to detect access points to determine the corresponding transactions. The extraction of the information on the generated AST is not easy to perform. This is the reason that the reduced model of the source code is mapped as logical facts using CLIPS [11], which is a productive development and delivery expert system tool which provides a complete environment for the construction of rule and/or object based expert systems. Thus, any reasoning regarding the target model is simplified to writing logical rules, and we stay still independent of the source code.

4) Inferring inheritance rules. In previous steps, we just analyzed the object-oriented model from a syntatic viewpoint. But we have to consider inheritance relationships that have influence in the computation of nP. Specifically, we have to consider those methods that are implemented in a class A, but from the inheritance viewpoint are inherited by all the (direct and indirect) subclasses of A.

From the previous step, only direct inheritance relationships are mapped as logical facts, i.e. if a class A is superclass of class B, and class B is superclass of class C, we have two logical facts: superclass(A,B) and superclass (B,C). However, there is no information regarding indirect (transitive) inheritance relationships. In our example, A is superclass of C. Firstly, we add logical rules that show this relationship.

Then, using the information generated for the transitive inheritance relationship, we now complete the logical model with new facts that show all possible receivers of a method m implemented in a class A. This means that if we have a method m implemented by the class A, each subclass B of A that do not implement or overwrite the cited method can answer it. Thus, we add a new logical fact that shows that class B can answer method m.

4) Detecting access points. The detection of the access points to build the transactions is designed with logical rules implemented in CLIPS. The rules are the following ones:

a. Class A has a method m as an access point, if m is defined in class A, class A implements the interface ActionListener, and the signature of m is void actionPerformed (java.awt.event.ActionEvent).
b. Class A which implements main(String[]) as static method.
c. Other rules for the specific implementation environment

5) Building Transactions using access points. The previous step identified the access points of the target program. We have to use them to build the transactions on the source code. As a first step, we identify which are the transactions in the different analyzed use cases. Once we have identified them, we can have different mapping strategies:

a. A transaction with a unique access point.
b. A transaction with multiple access points: For example, in a form with different fields, a piece of code is executed whenever the user fills in one. However, the transaction ends when all the fields were filled in.

Table 1. Class, Access point, Use Case and Transaction identified in a Traceability relationship

Class	Access point	UC	Actor's actions of UC T
atm.ATM	(performShutdown)	System Shutdown	turns off
atm.ATM	(performStartup)	System Startup	enter (Startup)
atm.Session	(performSession)	Session	insert, enter (Session)
atm.transaction.Deposit	(complete Transaction)	Deposit Transaction	Accept
atm.transaction.Deposit	(getSpecificsFromCustomer)	Deposit Transaction	Choose_D
atm.transaction.Inquiry	(complete Transaction)	Inquiry Transaction	Choose_I
atm.transaction.Inquiry	(getSpecificsFromCustomer)	Inquiry Transaction	Choose_I
atm.transaction.Transaction	(performTransaction)	Transaction	Choose_T
atm.transaction.Transfer	(complete Transaction)	Transfer Transaction	Choose_Tr
atm.transaction.Transfer	(getSpecificsFromCustomer)	Transfer Transaction	Choose_Tr
atm.transaction.Withdrawal	(complete Transaction)	Withdrawal Transaction	Choose_W
atm.transaction.Withdrawal	(getSpecificsFromCustomer)	Withdrawal Transaction	Choose_W
atm.transaction.Transaction	(performinvalid PIN)	Invalid Pin Extension	re-enter
ATMMain…	(actionPerformed)	Transaction not specified	
simulation.ATMPanel.72…09	(actionPerformed)	Transaction not specified	
simulation.BillsPanel.26… 84	(actionPerformed)	System Startup	enter (Startup)
simulation.CardPanel.61…43	(actionPerformed)	Session	Insert
simulation.LogPanel.45… 72	(actionPerformed)	Transaction not specified	
simulation.LogPanel.45…63	(actionPerformed)	Transaction not specified	
simulation.SimCardReader…	(actionPerformed)	Session	Insert
simulation.SimEnvelopeAc…	(actionPerformed)	Deposit Transaction	Accept
simulation.SimKeyboard…08	(actionPerformed)	Session, Deposit Trasa, Transfer Trasa	enter(Session),choose (all)
simulation.SimKeyboard…09	(actionPerformed)	Session, Deposit Trasa, Transfer Trasa	enter(Session),choose (all)
simulation.SimKeyboard…38	(actionPerformed)	Session, Deposit Trasa, Transfer Trasa	enter(Session),choose (all)
simulation.SimKeyboard…74	(actionPerformed)	Sessio, Deposit Trasa, Transfer Trasa	enter(Session),choose (all)
simulation.SimOperatorPanel.	(actionPerformed)	System Startup, System Shutdown	turns on, turns off
simulation.SimReceiptPrinter.	(actionPerformed)	Transaction not specified	

a. Several transactions with the same access point. A transaction can be an alternative path inside the code, that is executed starting from an access point, which is common to several transactions.

6) Computation of the nP of each identified transaction. Once we have identified the transactions, we compute the complexity of them based on the number of paths of the method calls, as said previously in this section.

3. CASE STUDY

To validate our approach, we use a specification of an ATM System [12] to validate the presented approach. We chose this case study because it is a middle-sized one, it was developed by third parties and we have the textual descriptions of the use cases and the Java source code. Table 1 shows the traceability between use cases, transactions identified on use case textual descriptions and the identified access points in code. Table 2 shows the use case names, the measured paths (*nP*) based on the use case textual descriptions and source code.

Table 2. Path measured on UC and Code

Use case name	nP (UC)	nP (Code)+ k
System Startup Use Case	2	18
System Shutdown Use Case	1	8
Session Use Case	4	81
Transaction Use Case	4	94
Withdrawal Trasaction Use Case	2	33
Deposit Trasaction Use Case	3	115
Transfer Trasaction Use Case	1	52
Inquiry Trasaction Use Case	1	17
Invalid Pin Extension	3	85

Analysis of the Results: From Table 1, we can observe that our hypothesis that there is a dependency between transactions at use case level and source code level is confirmed. Thus, we consider that the transaction is *traceable*. However, the traceability relationship between transactions at use case level and code level is not one to one, but it is zero to many. This means that a code transaction may be not defined at the use case level, and that a use case transaction may be decomposed in more than one transaction at the code level. But when analyzing the results, we do not keep traces between paths at use case level and at code level (as we do in transactions), because an alternative path is not identified as a requirement unit, but is defined by a principal path. Even when we could find traces between paths, so far the information is not relevant in our analysis.

Table 2 shows an important difference between nP (UC) and nP (Code), the value of nP(Code) are higher than the values of nP (UC) . There are also differences between use case transaction and code transactions. This result can be considered normal because the level of detail of developers is different when they work in requirements phase and programming phase.

Discussion. Regarding the methodology, the most difficult aspect was the identification of the access points in the source code to build the transactions, because it is based on rules that we have built. In this paper, we limit the number of rules, but analyzing other case studies, other rules can be added to refine the results. Even when we have worked with a case study implemented in Java, we have said previously that our approach is independent of the target language. In order to analyze an application implemented in another language, we have to use the corresponding source code model, other than Recoder, in the step 1, and adapt the rules for the access-point detection in step 4.

4. RELATED WORK

To our knowledge, there are no bibliographic references that deal with traceability in complexity metrics. However, several works show interesting aspects of traceability metrics. Due to the reduced space in the paper, we just mention briefly them

Pfleger et al. [13] cope with processing measurements because they are more difficult to track, as they often require traceability from one product or activity to another one. Antoniol et al. [14] presents a method to maintain traceability links between subsequent releases of a software system. Shepperd [15] investigates the various existing metrics for system component size. An alternative metric is proposed, based upon the traceability of functional requirements from a specification to design. Paul et al. [16] present approaches for current software metrics database environments to achieve efficient execution and management of large projects. They proposed a combination of critical metrics and analytical tools that can enable highly efficient and cost effective management of large and complex software projects.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an approach and our initial validation to detect and measure the complexity of traceable transactions from requirements (use cases) to code (source code) of a target software system. With our initial experiments, we have shown that our complexity metrics (nP and nT) are useful to evaluate if the requirements were implemented or not, and how they were implemented. Thus, we can offer developer a global view of the system in the complete software development process.

Even when the results are promising, there are still some future work. We want to analyze if there is a correlation between nP measured in use cases and in source code. We want also to test our approach in other object-oriented languages to see the adaptability issues in these cases.

6. ACKNOWLEDGMENTS

Our thanks to the Research Fund of School of Engineering of Austral University, which made this study possible.

7. REFERENCES

[1] Jacobson, I., Booch, G.and Rumbaugh J. 2003*The Unified Software Development Process*. Addison-Wesley.

[2] Booch, G, Jacobson, I. and Rumbauggh J. 1997 *The Unified Modeling Language User Guide*, Addison-Wesley.

[3] ISO/IEC 20926: 2003, Software engineering – IFPUG 4.1 Unadjusted functional size measurement method – Counting Practices Manual, International Organization for Standardization, Geneva.

[4] Karner, G. 1993. *Metrics for Objectory*. Diploma thesis, University of Linköping.

[5] Robiolo, Gabriela; Badano, Cristina; Orosco, Ricardo, 2009 Transactions and Paths: two use case based metrics which improve the early effort estimation, *Empirical Software Engineering and Measurement (ESEM)*

[6] Chidamber, S. and Kemerer, C. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6. IEEE Press.

[7] McCabe, T. 1976. A Complexity measurement, *IEEE Transactions on software Engineering*, Vol. SE-2, NO. 4. IEEE Press

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

[9] Watson, A.H. and Mac Cabe, T.J.. *1996. Structured Testing: Atesting Methodology using the Complejidad Ciclomática Metric*, National Institute of Standards and Technology, Gaithersburg, http://www.mccabe.com/pdf/nist235r.pdf

[10] http://recoder.sourceforge.net/index.html

[11] http://clipsrules.sourceforge.net/WhatIsCLIPS.html

[12] http://www.cs.gordon.edu/courses/cps122/ATMExample/index.html

[13] Pfleeger, SL., Jeffery, R., Curtis, B. and Kitchenham, B. 1997. Status report on software measurement. *IEEE software*

[14] Antoniol, G., Canfora, G., Casazza, G. and De Lucia, A. 2001 Maintaining traceability links during object-oriented software evolution. *Softw. Pract. Exper*. 2001; 31:331–355.

[15] Shepperd, M. and Ince D.1990 Multi-dimensional modelling and measurement of software designs.*ACM Annual Computer Science Conference, ACM annual conference on Cooperation.*

[16] Paul, R., Kunii, T., Shinagawa, Y. and Khan, M. Software 1999 Metrics Knowledge and Databases for Project Management. *IEEE Transactions on Knowledge and Data Engineering*, VOL. 11, NO. 1.