McC and Mc#: Unified C++ and C# Design Facts Extractors Tools

Petru Florin Mihancea, George Ganea, Ioana Verebi, Cristina Marinescu and Radu Marinescu
LOOSE Research Group
"Politehnica" University of Timişoara, Romania
lrg@cs.upt.ro

Abstract

In the last years, as object-oriented software systems have become more and more complex, the need of performing automatically reverse engineering upon such systems has significantly increased. It is well known that one step toward a research infrastructure accelerating the progress of reverse engineering is the creation of an intermediate representation of software systems. In the current demonstration we present an unified structure for representing object-oriented systems written in C++ and C#, together with the corresponding model capturing tools. As a result, we can uniformly analyze C++ and C# systems. Moreover, we have integrated the tools in the iPlasma reengineering infrastructure which permits us to obtain easily valuable information for a reverse engineering process.

1. Introduction

In the last years, as object-oriented systems have become more and more complex, the need of powerful techniques to recover the understanding and to assess their design quality has significantly increased. As stated in [13], one step towards a research infrastructure accelerating the progress of these *reverse engineering* [1] means is the creation of an intermediate representation of software systems.

Usually, this intermediate representation is a *model* of the analyzed system. It contains specific information (*i.e.*, design information) extracted from the source code and conforms to a *meta-model*. At this time, there is a plethora of such meta-models defined by different research groups for various reverse engineering purposes and/or dedicated to particular programming languages of interest.

Such a "plethora" of meta-models has also appeared in our LOOSE Research Group. First, a meta-model for C++ systems called *TableGen* [9] has been defined and implemented using a relational paradigm (a SQL database). Later, due to the fact that Java become more and more widely used, the need of reverse engineering Java systems has also increased. Consequently, another meta-model called *MEMOJ* for representing systems written in this programming language has been defined and implemented in Java.

At that moment, developing a new design analysis required an additional overhead due to the need of actually having two implementations: one in the relational paradigm (for C++ systems) and another one in Java (for Java systems). Moreover, *TableGen* was the subject of different flaws [11].

In order to remove these inconveniences a new *unified* meta-model called *MEMORIA* [12] for Java and C++ systems has been defined. At the same time a new design facts extractor tool has been created (*McC*) which produces a *MEMORIA* compatible representation of a C++ system.

Furthermore, the continuos growing of software systems implemented using C# language increases the need of a meta-model for representing design information for these systems too. This also made us create a design fact extractor for C# systems also compatible with the *MEMORIA* meta-model.

In this tool demo we present two design facts extractors tools (*McC* - for C++ code, respectively *Mc#* - for C# code) together with the unified way the design artifacts are organized. Next we present some reverse engineering analyzes that can be uniformly performed by manipulating the information extracted by these tools in our reengineering environment called *iPlasma* [8].

2. Unified C++ and C# Facts Extractors Tools

2.1. McC – a C++ Facts Extractor Tool

McC (Model Capture for C++) is a reverse engineering instrument which extracts *design* information from C++ software systems. In essence, it receives as input a folder containing the source code of the analyzed system and produces as output a set of ASCII tables containing the extracted information.

2.1.1. Design Facts. Presenting all the design information extracted by McC is next to impossible due to the space lim-

0-7695-3078-8/08 $25.00 © 2008 IEEE
DOI 10.1109/SYNASC.2007.13

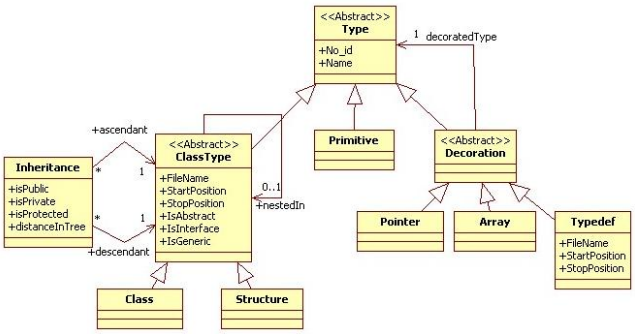

Figure 1. Type and Inheritance

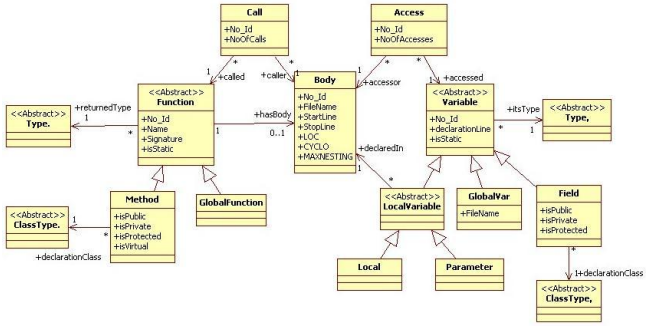

Figure 2. Function, Body, Variable and Others

itations. As a result, we decided to present only a partial view of this information together with a set of UML class diagrams specifying the way the design facts are structured.

Type Table. The entries of this table are actually serialized instances of the concrete *Type* classes from Figure 1. McC captures the used *Privitive* types from an analyzed system and, more important for object-oriented design analysis, each declared *Class* and *Structure*. For the latter entities, McC stores information like theirs *Names*, the *FileNames* (and the position in those files) where they are declared, checks to see if they are abstract (*i.e.*, contain an abstract method) or, even more, if they are pure interfaces (*i.e.*, a pure abstract class), etc. McC also captures relations between *ClassTypes* such as the nesting of classes and also models *Pointers*, *Arrays* and *Typedef* types definitions as distinct types using a *Decorator* design pattern [5].

Inheritance Table. This table contains serialized instances of the association class *Inheritance* from Figure 1. Such an object represents an ascendant-descendat relation between two *ClassTypes*. The relation is characterized by the inheritance visibility attribute (*e.g.*, public) when it is direct (*i.e.*, the descendant is a direct subclass of the ascendant) and by the length of the path between the ascendant and the descendant in the inheritance lattice.

Body Table. Each entry from this table is an instance of the *Body* class presented in Figure 2. Such an object corresponds to the body of a function and it is characterized by the *FileName* where it is defined (and the position in the file) and by a set of primary design metrics also computed by McC : *Cyclomatic Number* [10], the *Maximum Nesting Level* [7], the *Number of Lines of Code*, etc.

Function Table. This tables contains information regarding the declared functions of the analyzed system. In essence, for any concrete kind of function (*e.g.*, *Global* or *Method*) McC records in this table the *Name* of the function, its *Signature*, the presence of the static modifier, the unique identifier of the *Type* object representing its returned type and a reference to its body (if exists). For a *Method* supplementary information is extracted such as its access specifier, the *ClassType* object that contains the *Method* and the presence of the virtual declaration.

Variable Table. Information regarding different kinds of variables used in the analyzed system are stored in this table. For any kind of variable McC records here the *DeclarationLine*, the presence of the static modifier and the unique identifier of the *Type* object representing its declared type. For a *LocalVariable* (*i.e.*, classical local variable or parameter) McC also identifies the *Body* object that contains that variable. For a *Field* the access rights are identified and also the *ClassType* object that contains that *Field*.

Call and Access Tables. Instances of the association classes *Call* and *Access* represent entries of these tables. A *Call* entry specifies that its associated *Body* object contains a call to the associated *Function*. In other words the *Call Table* represents a static call-graph (it does not consider dynamic dispatched calls). A *Call* entry is also characterized by the number of distinct invocations to the *Function* found in the *Body*. In a similar way an *Access* object specifies that its associated *Body* object reads or writes the associated *Variable*.

Other information. As we already mentioned, McC also extracts other kinds of design information which is not presented here. This includes information regarding template parameters, relations between template parameters and concrete types used to instantiate it, information related to namespaces, etc. The complete description of the tables can be found in [11].

2.1.2. Implementation Details. McC has been implemented in Visual Studio 6.0 and it is based on the Telelogic's FAST parsing library [2]. It makes use of some special facilities of this library such as the ability to configure it for different C++ dialects via configuration

files. McC has been also designed to be easy to extend. Based on a couple of design patterns (*e.g.*, Template Method, Chain of Responsability [5]) it is simple to modify the tool to extract new design facts from a C++ system in the form of new columns in the already defined tables and even in the form of new tables.

2.2. Mc# – a C# Facts Extractor Tool

As McC, Mc# (Model Capture for C#), receives as input a folder containing the source code of the analyzed system and produces as output a set of ASCII tables containing the model of the system.

2.2.1. Design Facts. Due to the fact that most existing design entities are common for C++ and C# (*e.g.*, classes, methods, attributes), it is possible to have an unified structure (presented in Section 2.1.1) for the tables produced by Mc# and McC. Nevertheless, some differences between the two languages exist – for example, they have different access modifiers (C# has also *internal* and *protected internal* access modifiers). At this time, Mc# considers that access modifiers have, like in C++, only three types (public, protected and private). Entities declared as internal or protected internal are considered to have the default C# access (*e.g.*, private). Here we must emphasize that minor details like this one are not extremely relevant from design analyses point of view.

2.2.2. Implementation Details. Mc# has been implemented in Visual Studio 2003 and it is based on the Metaspec's [1] parsing library.

3. Evaluation

3.1. Performances

McC has been used as the model extractor tool for many research case studies regarding the design of C++ systems [7]. Moreover, it has been also used in several industrial case studies for prestigious companies from *Timisoara (Romania)* and *Switzerland*. In order to emphasize its performances we present in Table 1 some concrete analysis conditions. The tests have been performed on a computer having an Intel P4 2.8GHz processor, 1024 RAM and running Windows Xp.

At the moment, Mc# has been used for extracting facts from C# open-source software systems, most of them being found at *www.sourceforge.net*. Table 2 presents some results obtained by parsing three systems using an Intel Core 2 Duo processor and 1024 RAM.

1 A free trial version can be found at http://www.csharpparser.com/C#

Infos ans Systems	System1	System2	Mozilla
Size	∼5 Mb	∼30 Mb	∼80Mb
Types	2.050	12.855	24.091
Functions	3.806	69.609	82.908
Variables	11.323	90.995	256.905
Time	∼5min	∼13h	∼28h

Table 1. McC Performances

Infos ans Systems	System1	System2	System3
Size	∼1.2 Mb	∼5.5 Mb	∼17.5Mb
Types	245	1.215	3.601
Functions	637	4.419	17.392
Variables	2.442	12.475	56.365
Time	∼1min	∼13min	∼2h

Table 2. Mc# Performances

3.2. Applications

Both tools are part of the *iPlasma* reengineering environment [8]. In order to make use of the entire analysis power offered by the *INSIDER* front-end of this platform we have also built a table importer. In essence, this additional "tool" transforms the ASCII tables produced by McC and Mc# into an instance of the *MEMORIA* meta-model. As a result, we can analyze in *INSIDER* from the design point of view Java, C++ and C# systems in an *uniform* way (*i.e.*, a design analysis is implemented only once). Figure 3 shows the main components of the *iPlasma* environment and the way the design information extracted by McC and Mc# can be further exploited.

As we can observe, several types of design analyses can

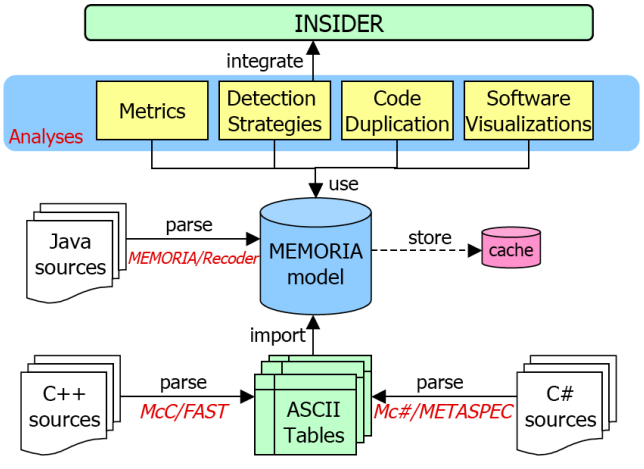

Figure 3. iPlasma Environment

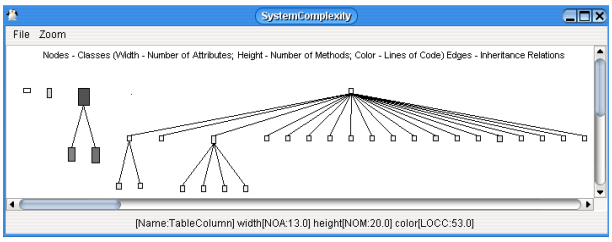

Figure 4. System Complexity View Example

be performed on C++ and C# systems such as design metrics computation, the detection of design problems using metrics-based rules or the generation of different software views [7]. Next, we are going to present some examples.

Metrics. As we have already mentioned, McC and Mc# provides us with a set of primary metrics. For example, knowing the values of the *Number of Lines of Code* for the methods of an inspected system provides us with the methods which are the subject of an obfuscated understanding, according to the Fowler's bad smell *Long Method* [4]. Beside primary metrics, we have implemented on the top of the MEMORIA meta-model different compound metrics like *Access to Foreign Data* and *Changing Classes* [7].

Metrics-based rules for detecting design problems. This type of analyses (*i.e.*, detection strategies [7]) helps us to identify those entities affected by different design problems. Using this techniques we can identify in the models generated by McC and Mc# design flaws such as *Data Class* [4], *Feature Envy* [4], *Duplicated Code* [4], etc.

Software Visualizations. Based on the design information extracted by McC and Mc# we can generate in *INSIDER* different software views such as *System Complexity* or *Class Blueprint* [7]. In Figure 4 we present the *System Complexity View* of a C++ system, a visualization that can be used to locate the important hierarchies from a system or exceptional classes in terms of their size (*e.g.*, many lines of code).

4. Related Work

The most related approach we have found is *Columbus* [3]. *Columbus* is a reverse engineering framework on top of which different reverse engineering and reengineering analyses can be implemented for C++ software systems. Together with this framework, the authors introduce a schema for representing C++ source code. In [6] an infrastructure to support interoperability between various reengineering tools is introduced. The authors propose an API through which information about the declarations (*i.e.*, classes, functions and variables) from a C++ system can be extracted and a hierarchy of canonical schemas to represent various graph structures needed during different C++ source code analyses. In this tool demo, we present a *MEMORIA* compatible *unified* representation for *design* information extracted from C++ and C# software systems. We also present two tools that extract design artifacts according to this structure from C++ (*McC*) respectively from C# (*Mc#*) programs. Last but not least, we briefly show how the extracted design facts can be *uniformly* manipulated in *INSIDER* (the front-end of our *iPlasma* reengineering environment) exemplifying by different concrete design analyses.

Acknowledgments. A part of this work is supported by the Romanian Ministry of Education and Research under Project CEEX(5880/18.09.2006). Petru Mihancea thanks for the financial support offered by CNCSIS through the research grant T3/46/GR/11.05.2007.

References

[1] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. In Robert S. Arnold, editor, *Software Reengineering*, pages 54–58. IEEE Computer Society Press, 1992.

[2] SEMA FAST Parser. Internal Programmer's Manual 2001.

[3] Rudolf Ferenc, Ferenc Magyar, Arpad Beszedes, Akos Kiss, and Mikko Tarkiainen. Columbus - reverse engineering tool and schema for C++. In *Proceedings of ICSM'02*. IEEE Computer Society, 2002.

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[6] N.A. Kraft, B.A. Malloy, and J.F. Power. Towards an infrastructure to support interoperability in reverse engineering. In *Proceedings of WCRE'05*. IEEE Computer Society, 2005.

[7] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[8] Cristina Marinescu, Radu Marinescu, Petru Florin Mihancea, Daniel Ratiu, and Richard Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *ICSM (Industrial and Tool Volume)*, pages 77–80, 2005.

[9] Radu Marinescu. TableGen. A tool for extracting design information from C++ code. Technical report, Politehnica University of Timisoara, 2000.

[10] T.J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[11] Petru Florin Mihancea. The extraction of detailed design information from C++ software systems. Master's thesis, Faculty of Automatics and Computer Science, "Politehnica" University of Timişoara, 2004.

[12] Daniel Raţiu. Memoria : A Unified Meta-Model for Java and C++. Master's thesis, Faculty of Automatics and Computer Science, "Politehnica" University of Timişoara, 2004.

[13] S. Rugaber and L.M. Wills. Creating a research infrastructure for reengineering. In *Proc. IEEE Working Conference on Reverse Engineering*, 1996.