Analysis and Manipulation of Distributed Multi-Language Software Code

L. Deruelle, N. Melab, M. Bouneffa and H. Basson
Laboratoire d'Informatique du Littoral
Université du Littoral
50, rue Ferdinand Buisson, BP 719.
62228, Calais Cedex, France
e-mail : {deruelle, melab, bouneffa, basson}@lil.univ-littoral.fr

Abstract

In this paper, we propose a formal model and a platform to deal with distributed multi-language software analysis. These provide a graph representation of the software codes (source codes and byte-codes), a change propagation process based on graphs rewriting, and an automatic profiling tool to measure the contribution of any component to the global performance of the software. The program codes are structured by a multi-graph in which the nodes represent the software components linked by edges representing the meaningful relationships. The software components and their relationships are extracted from the byte-code files, using *mocha decompiler tool*, and from the source codes files, using *Javacc tool*. *Javacc* allows to generate parsers, based on grammars specifications files, which include features to produce an XML (eXtensible Markup Language) representation of the software components. Furthermore, a graph of the software components is constructed on the top of the XML files, providing programs analysis. This is implemented by an integrated platform including the *mocha* decompiler, a multi-language parsing tool, a software change management module, and a profiling tool.

Keywords: *Source code analysis, Byte-code, Profiling, Change Propagation, Decompiling.*

1 Introduction

The software can be viewed as a multi-graph in which the nodes represent components like functions or classes and the edges are the meaningful relationships like function calls or classes inheritance. A major part of the software maintenance is the change propagation [1, 2, 18]. Indeed, every change even very small applied on a code component can have critical side effects if it is not seriously analyzed. The impact can be determined by a change propagation process, which propagates it from component to component through the relationships. For instance, if one has to change the name or type of a parameter of a given function he/she has to change all the code components calling that function. We say that the change of the function has an impact on the other components. This is propagated through the calling relationship. In brief, change propagation analysis is based on a software code structural model and a change propagation process [18].

The model represent the code components

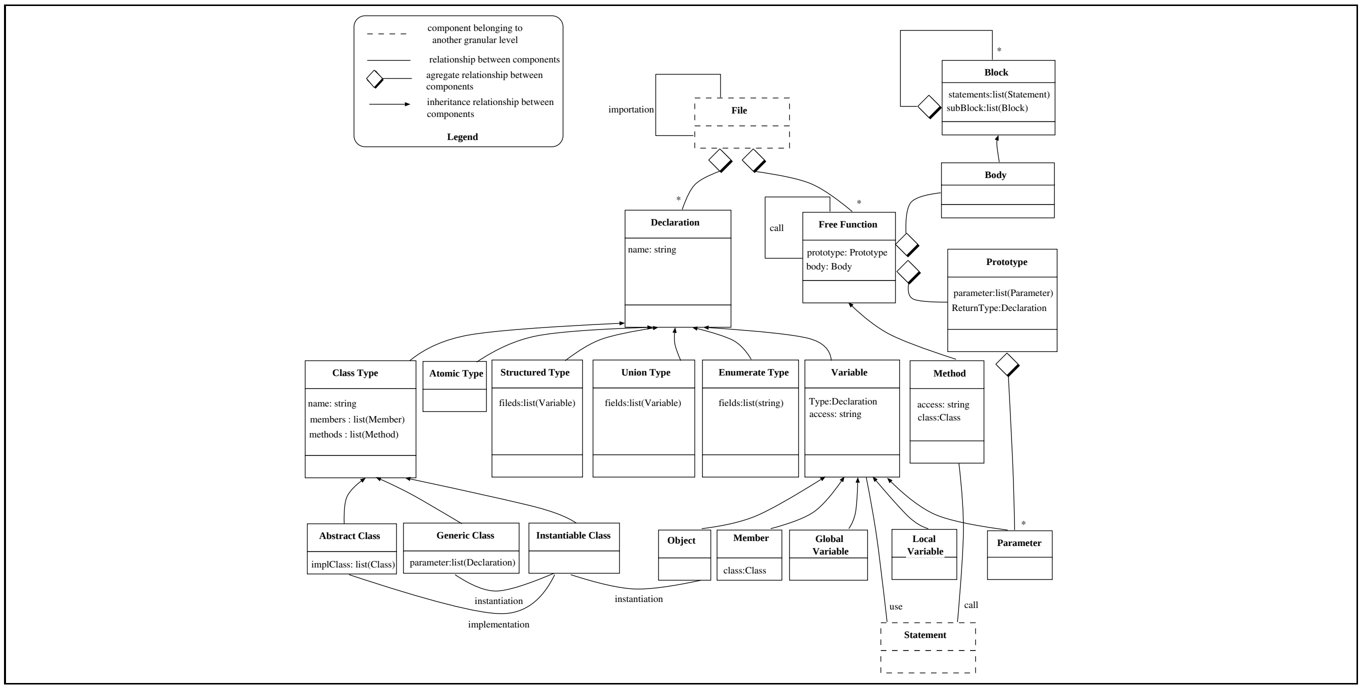


Figure 1: UML-based representation of SCSM

and their relationships. In [13], we proposed such a model, named *Source Code Structural Model* (*SCSM*). We applied it for profiling of object-oriented code. In [4], we extended it to deal with database schemas, and experimented it on schema change propagation. A major advantage of the model is the scalability making it able to deal with a large variety of programming languages, database management systems and middlewares. A platform implements the model features and provides a multi-language source code analyzer, a change propagation process and an instrumentation-based profiling technique. The source code analyzer includes parsers generated by means of *Javacc tool*. Each parser is based on a language specific grammar and provides an XML representation of the code. The XML representation contains a set of software components and their relationships, structured in a multi-graph, that is used to build an object-oriented components repository. The change propagation process and the profiling technique use the multi-graph as inputs and provide respectively the impact of changing a component or its performance measurement.

This paper is organized as follows: in Section 2, we propose a description of the model for distributed multi-language software code. Section 3 describes the multi-language code parsing process. Section 4 gives two applications: 1) the change propagation analysis performed on a bank application, 2) the instrumentation of the Objectstore Query package. In section 5, concluding remarks are drawn.

2 An Abstract representation Model

2.1 The Basic Model: SCSM

SCSM describes a general abstract structure of the source code of a software application composed of programs like Perl or PHP scripts, Java or C/C++ source codes and relational or object-oriented database schemas. It includes two kinds of elements, as shown in figure 1 : the *software components* and the *relationships* between them. The software components can be *coarse-grained* like *applications*, *files* or *libraries*. They can be of *medium granularity* such as *generic classes*, *effective classes*, *attributes*, *class methods*, *objects*, *free functions/procedures* or *code blocks*. They also can be *fine-grained* such as *statements*, *expressions*, *control structures* or *individual symbols*.

On the other hand, all these software components are linked by basic (binary and direct) relationships. The non-exhaustive list of identified relationships is the following: *resource importation*, *calling*, *inheritance*, *composition*, *friendship* or *instantiation*.

To make the model usable for effective experimentations we implemented its features with Java [13] on top of the ObjectStore PSE/PRO database system [5].

2.2 DSCSM : an extension of SCSM to distributed software

The basic model have first been extended to deal with distributed components. For this we have included the CORBA features into the model.

2.2.1 A brief presentation of CORBA

The Common Object Request Broker Architecture (CORBA) is a middleware specified by the Object Management Group (OMG). It allows the integration and the interoperability of legacy applications. It includes an Interface Definition Language (IDL) and a communication infrastructure named Object Request Broker (ORB). IDL allows the programmer to specify with an OMG-IDL file a contract (set of type definitions and interfaces) between a server (supplier) of objects and a client (consumer). Once the contract (IDL file) is specified, it is projected to an implementation language. For example, if this language is Java the different IDL interfaces are translated into Java interfaces. With C++, IDL interfaces are projected to classes. Finally, the user has to implement the interfaces and to write the server and client codes. On the other hand, the ORB ensures the communication between the client and the objects of the server resulting from the spreading out of the interfaces.

2.2.2 Extensions of SCSM to CORBA applications

OMG-IDL can be viewed as a small extension of a sub-part of C++ with some components of Java like *interface* and other components like *sequence*. A sequence is a list of homogeneous elements whose size is unknown. As our model can be used for C++ and Java software its extension to IDL is very easy. Indeed, in order to take into account CORBA distributed objects, the *SCSM* is extended with the *IDL file* and its components such as *sequence*. Moreover, *SCSM* is enriched with the relationships between the OMG-IDL components. Furthermore, an important relationship is considered, it is called *projection*. It links the *IDL* interfaces to the different classes resulting from the projection of the *IDL file* to an implementation language (Java for example).

The next section describes the extraction of the set of software components from software code, using a multi-language parsing tool.

3 A Multi-Language Code Parsing : a prototype implementation

As it is shown in Figure 2, the platform includes four major elements: *a multi-*

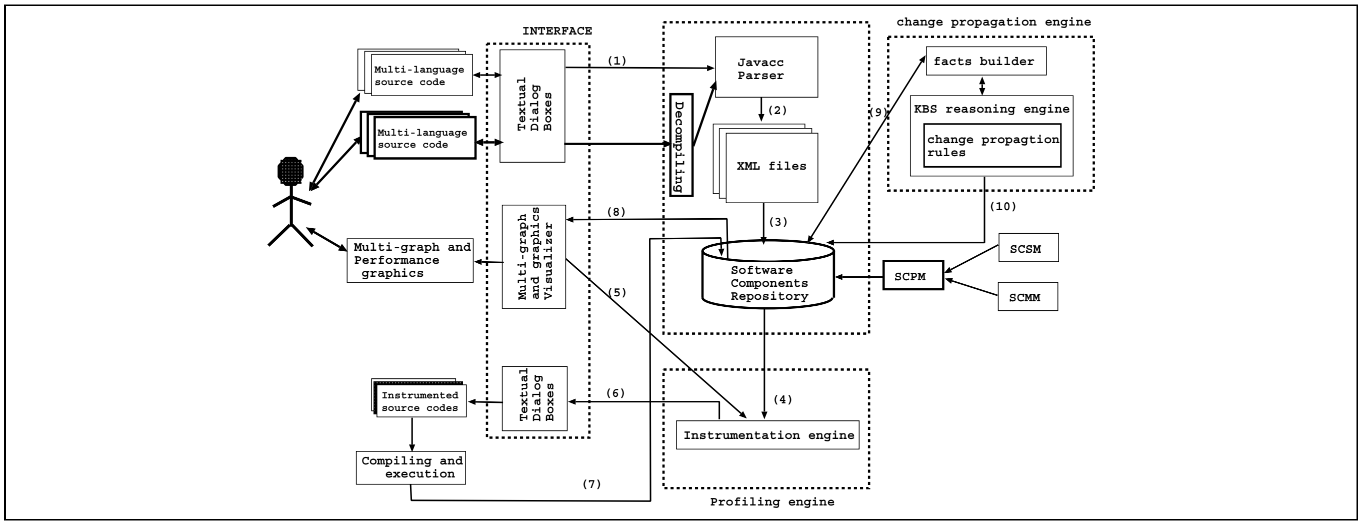


Figure 2: The global architecture of the prototype platform

language parser, *a profiling engine*, *a change propagation engine*, and a *graphical interface*. The first element parses using *javacc* [17] the multi-language (C, C++, Java, Cobol, MySQL, PHP, etc) source code of any input software. The input can also be a byte-code. In that case, a decompiling step is necessary, that is why our platform includes a byte-code decompiler (*mocha*). From the input or resulting source code, the platform generates (2) a set of XML-based files that describe the components and the relationships between them, and represent an instance of *SCSM*. The instance is then stored (3), as a multi-graph including the new components and relationships, in a *Software Components Repository* which is an ObjectStore database. The *profiling engine* allows to select a software component form the multi-graph (4)(5) and instrument it (6), by inserting automatically code fragments, to collect metrics value (7). The *change propagation engine* is based on the Java Expert System Shell (JESS) and propagates a change performed on a component to the others (10). The propagation is based on *change propagation rules*, which are fired by inserting facts, provided by the *facts builder*. The *facts builder* inserts facts, that represents the software components and the relationships stored in the repository (9). The *graphical interface* provides the user interaction to visualize metric graphs (8) and change propagation results.

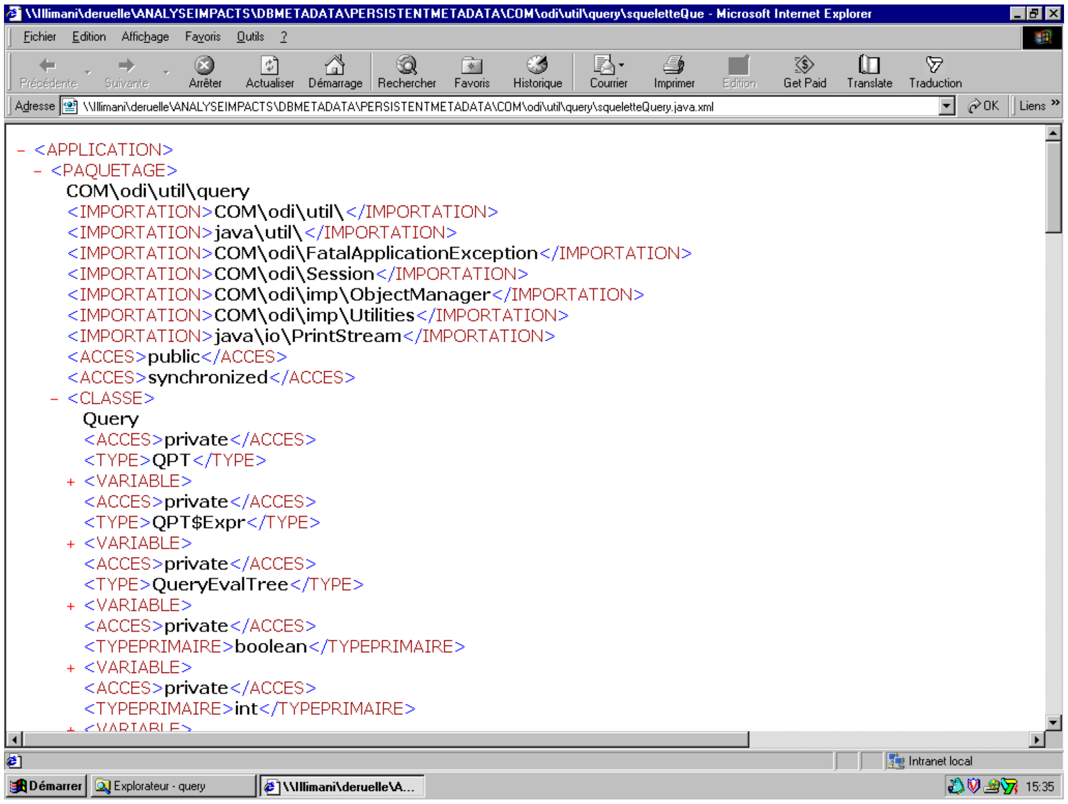
Figure 3 illustrates the XML representation of the software. The XML files contain software components and their relationships, stored into persistent ObjectStore objects as a multi-graph as illustrated in Figure 6.

The following section shows the application of the model for two main problems : *the change impact propagation* and *the software profiling.*

4 Applications

4.1 The Expert System-based Change Propagation Process

The change propagation process refers to the process of actually carrying out a set of initial modifications to the software components, and to re-establish the software consistency, by making a set of estimated consequent changes [10]. This process would involve ad-


\\Illimani\deruelle\ANALYSEIMPACTS\DBMETADATA\PERSISTENTMETADATA\COM\odi\util\query\squeletteQue - Microsoft Internet Explorer
Fichier Edition Affichage Favoris Outils ?
Précédente Suivante Arrêter Actualiser Démarrage Rechercher Favoris Historique Courrier Imprimer Edition Get Paid Translate Traduction
Adresse \\Illimani\deruelle\ANALYSEIMPACTS\DBMETADATA\PERSISTENTMETADATA\COM\odi\util\query\squeletteQuery.java.xml
OK Liens
Intranet local
Démarrer Explorateur - query \\Illimani\deruelle\A...
15:35

```
- <APPLICATION>
  - <PAQUETAGE>
      COM\odi\util\query
      <IMPORTATION>COM\odi\util\</IMPORTATION>
      <IMPORTATION>java\util\</IMPORTATION>
      <IMPORTATION>COM\odi\FatalApplicationException</IMPORTATION>
      <IMPORTATION>COM\odi\Session</IMPORTATION>
      <IMPORTATION>COM\odi\imp\ObjectManager</IMPORTATION>
      <IMPORTATION>COM\odi\imp\Utilities</IMPORTATION>
      <IMPORTATION>java\io\PrintStream</IMPORTATION>
      <ACCES>public</ACCES>
      <ACCES>synchronized</ACCES>
    - <CLASSE>
        Query
        <ACCES>private</ACCES>
        <TYPE>QPT</TYPE>
      + <VARIABLE>
        <ACCES>private</ACCES>
        <TYPE>QPT$Expr</TYPE>
      + <VARIABLE>
        <ACCES>private</ACCES>
        <TYPE>QueryEvalTree</TYPE>
      + <VARIABLE>
        <ACCES>private</ACCES>
        <TYPEPRIMAIRE>boolean</TYPEPRIMAIRE>
      + <VARIABLE>
        <ACCES>private</ACCES>
        <TYPEPRIMAIRE>int</TYPEPRIMAIRE>
      + <VARIABLE>
```

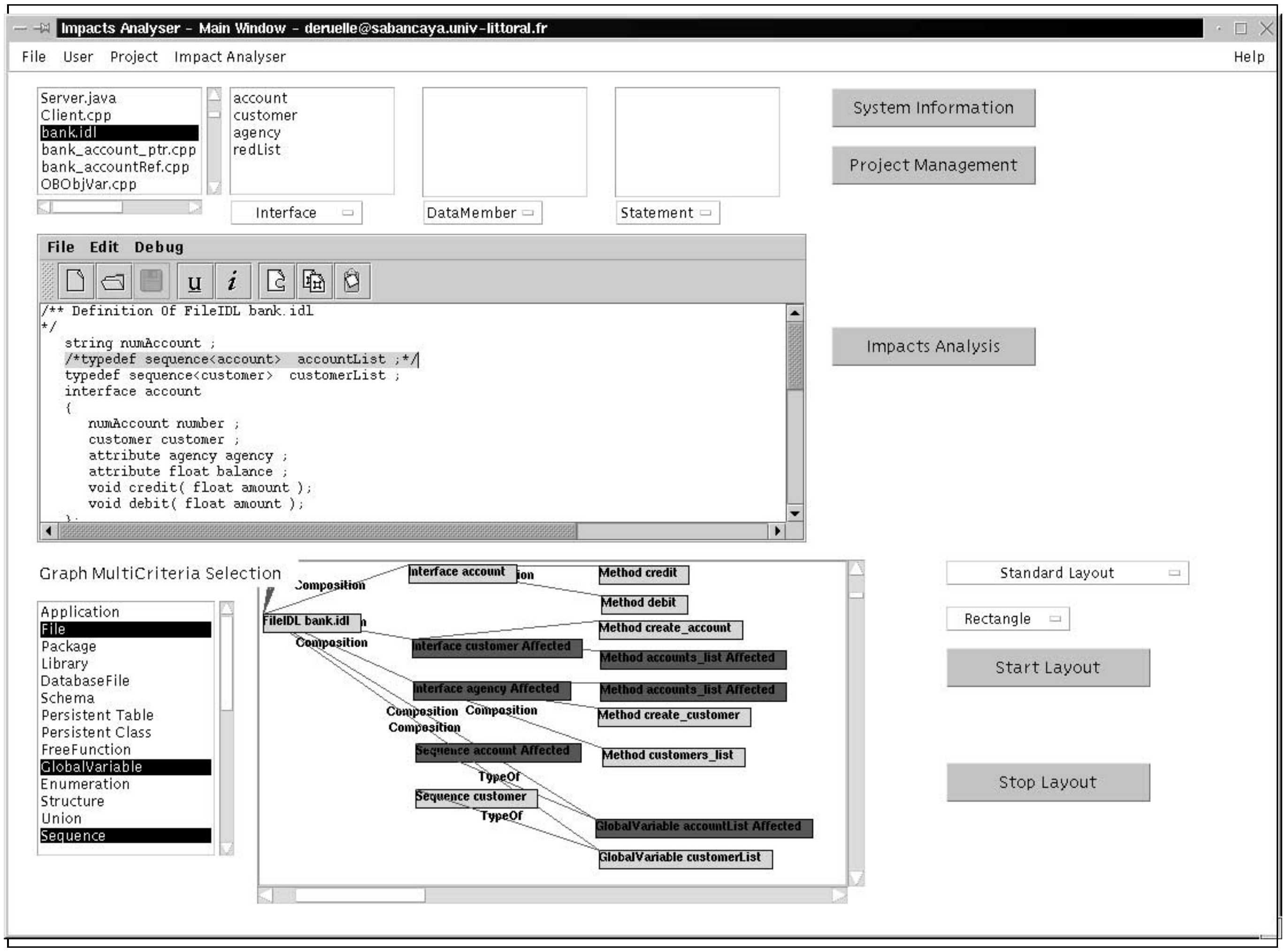
Figure 3: A snapshot of the XML representation of the software components

vising the user with the software components to be changed and the types of the changes. V. Rajlich has proposed an algorithm based on graph rewriting, called *Change-and-Fix*[18]. This algorithm considers a software as a dependency graph whose nodes and edges represent respectively the components and their relationships. For each component having been changed, the process starts by marking its neighboring nodes that must be changed. Then, the process is applied on the marked neighborhood. The process is repeated until no node is marked in the graph.

Two problems arise with the *Change-and-Fix* algorithm :

- The nodes of the graph are visited several times. An infinite process may occur when a change is propagated in a cycle (loops).
- When a node is affected by the change, either all its neighbors are marked or no one of them is marked. This is a consequence of considering all the relationship types as a unique one, namely dependency relationship. However, if we take into account the semantics of the different relationships we can refine the impact propagation to the direct neighbors by marking those really affected by the change.

In order to avoid the problems quoted above, we integrated an Expert System to the $Change - and - Fix$ algorithm. The new version is presented below. The facts of the Expert System are software components and their marking for change. The rules of the Expert System are the change propagation rules. The idea of our approach is to insert components to be changed and their marked neigh-



Server.java
Client.cpp
bank.idl
bank_account_ptr.cpp
bank_accountRef.cpp
OBObjVar.cpp

account
customer
agency
redList

```
/** Definition Of FileIDL bank.idl
*/
   string numAccount ;
   /*typedef sequence<account>  accountList ;*/
   typedef sequence<customer>  customerList ;
   interface account
   {
      numAccount number ;
      customer customer ;
      attribute agency agency ;
      attribute float balance ;
      void credit( float amount );
      void debit( float amount );
```

Application
File
Package
Library
DatabaseFile
Schema
Persistent Table
Persistent Class
FreeFunction
GlobalVariable
Enumeration
Structure
Union
Sequence

Figure 4: A snapshot of the impact propagation process

bors in the knowledge base (KB) (line (4)). Previously visited components may not be inserted again in the KB. Therefore, they can not be explored another time. In the algorithm, ($FireRules$) designates the set of propagation rules that can be fired.

```
(1) Given a consistent program represented by a graph $G = (V, E)$;
(2) Select $a \in V$;
(3) $Change(ModificationType, a)$;
(4) insert fact (ModificationType, $a$);
(5) $G'_a = (V'_a, E'_a)$ is the graph obtained by changing $a$;
(6) do {
(7)    do {
(8)       Select a rule $r$ in $FireRules$;
(9)       Trigger $r \cdot actionSet$;
(10)    }
(11)    while (FireRules $\neq \emptyset$);
(12)    Select a $\in$ (mark($G$));
(13)    $Change(ModificationType, a)$;
(14)    insert fact (ModificationType, $a$);
(15)    $G'_a = (V'_a, E'_a)$;
(16) } while ($mark(G) \neq \emptyset$);
```

4.1.1 The scenario

We consider here the experimentation of our process to deal with the change of a bank application. This application includes two distributed database applications communicating by means of CORBA. The CORBA-IDL specification of the application is the following:

```
module bank {
  interface customer;
  interface account;
  ...

  typedef sequence<account> accountList;

  interface account {
    ...
  };

  interface customer {
    ...
   accountList accounts_list ();
  };
  ...
};
```

Let us now consider the following change: deletion of the sequence *accountList* (becomes a comment in Figure 4) in IDL file. To explain how our Change-and-Fix/ES process determines the impact of the performed change, let us consider three levels in the application :

- The specification level : it includes the components of the IDL file containing the contract between a server of objects and a client.
- The intermediate level : as it is indicated above, once the IDL file is written it must be projected (compiled) to an implementation language. Each interface is projected to a set of files : the interface, the skeleton, the stub, etc. The IDL file is linked to this set of file s by the *projection* relationship. Therefore, the change of the IDL file is propagated to the projected files through that relationship. The intermediate level includes all the files resulting from the projection operation.
- The implementation level : it includes the different files containing the generated interfaces, their skeleton and stubs and their implementations.

As our platform is able to manage the change impact of C, C++ and Java, it is able to do it at the implementation level. For the scenario, as the interface *customer* is affected its following implementation is also affected by the change. Figure 4 illustrates the impact of the performed change determined by our platform. The components affected by the change have a black color and are marked by the term "*Affected*".

The following sections shows the second application of our platform that is the implementation of an automatic profiling technique.

4.2 Instrumentation-based Profiling

In [15], we identified three instrumentation levels : application, system and architecture. System-level instrumentation is done by some dedicated monitoring processes (e.g.. ALBA [16]) or by the operating system [12]. Architecture-level instrumentation [3] integrates into machines special monitoring hardware components (e.g. programmable counters). Application-level instrumentation [11, 8, 9, 7] can be: *manual*, *semi-automatic* or *automatic*. Manual instrumentation consists, of manually inserting code fragments into the original source code in order to evaluate profiling metrics (e.g. consumed CPU time) on code components (e.g. procedure). Semi-automatic approaches like that reported in the JEWEL tool [11] consist of *automatically* extracting the structure of the source code, and allowing the user to access the different components for example

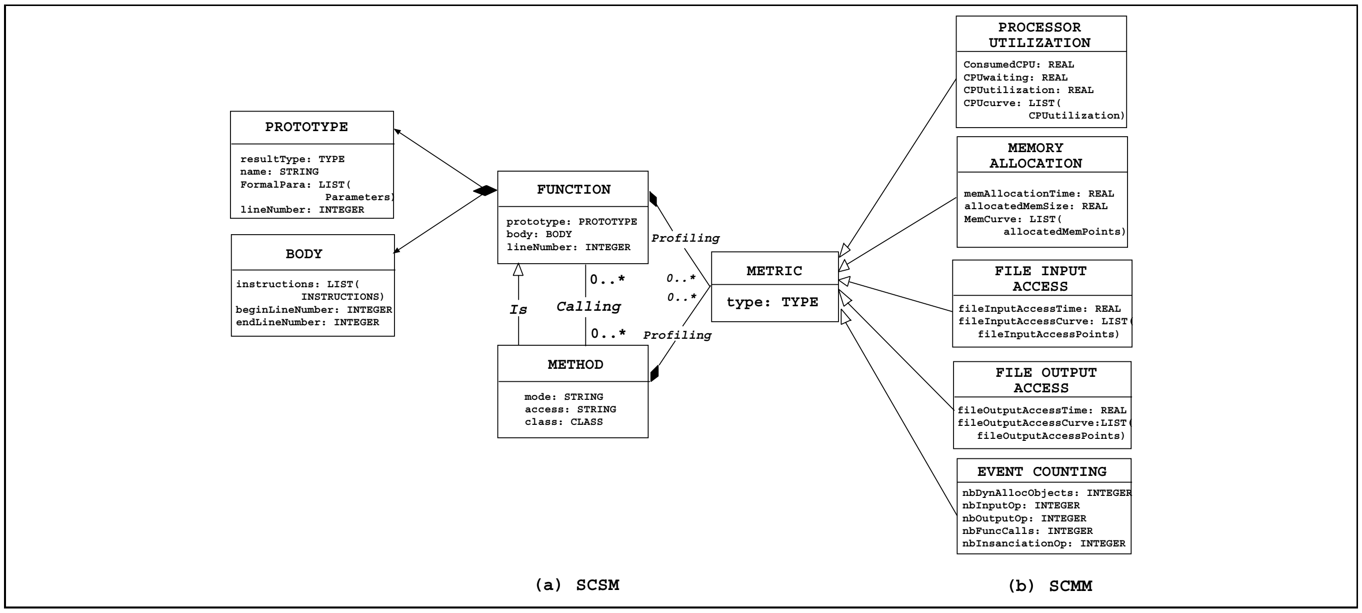


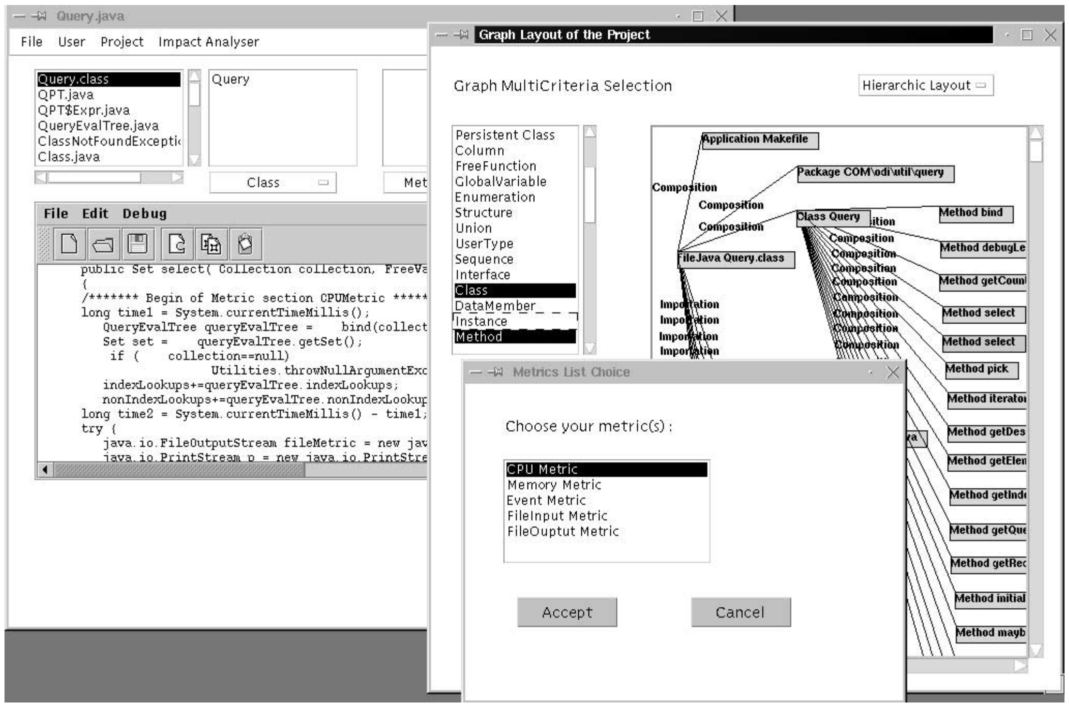
Figure 5: A partial UML-based representation of *SCPM*

through an interface. Then, he/she can choose the set of components to *manually instrument* and automatically generate the instrumented source code. With automatic techniques the user has not the burden of instrumenting his/her application. In previous works [13], we reported a source-to-source automatic application-level instrumentation approach of object-oriented code. The technique has been extended to distributed object programs in [14]. In this paper, we extend the work so that it deals with distributed byte-code programs. The approach is based on a profiling model, which defines associations between code components and metrics. The instrumentation can be viewed as an instantiation of these associations. The next section describes the model.

4.2.1 The Profiling Model

The profiling model, namely Software Components Profiling Model (*SCPM*), is a weak coupling between two basic models: a Software Components Structural Model (*SCSM*) and a Software Components Metrics Model (*SCMM*). On the other hand, as it is illustrated in Figure 5(b), *SCMM* contains five categories of metrics related to the resource consuming and event counting. These categories are the following: the processor utilization, the memory allocation, the file input access, the file output access and the event counting. Each category includes a non-exhaustive list of basic metrics. For example, the *processor utilization* category includes the elementary following metrics: the CPU time consumed by a component, its CPU waiting time, its CPU utilization time (the sum of the two previous times), its CPU utilization curve. The latter illustrates the processor activity associated with the execution of a given individual component.

All the software metrics cited above are associated with the following components of *SCSM*: application, object, code block, free function, class method and class. A file is associated only with the metrics related to the file input access, the file output access and the event counting. All the defined metrics



```
      public Set select( Collection collection, FreeVa
      {
      /******* Begin of Metric section CPUMetric *****
      long time1 = System.currentTimeMillis();
         QueryEvalTree queryEvalTree =    bind(collect
         Set set =    queryEvalTree.getSet();
          if (    collection==null)
                       Utilities.throwNullArgumentExc
         indexLookups+=queryEvalTree.indexLookups;
         nonIndexLookups+=queryEvalTree.nonIndexLookup
      long time2 = System.currentTimeMillis() - time1;
      try {
         java.io.FileOutputStream fileMetric = new jav
         java.io.PrintStream p = new java.io.PrintStre
```

Figure 6: Profiling model instantiation and the byte-code instrumentation result

are associated with the two components *function* and *class method*. Thus, one or several of these metrics can be instantiated (with persistent objects) into a Components Repository when the profiling data is collected at execution time.

Moreover, metrics have the polymorphism property. For example, the CPU utilization time is defined on the *free function* component as the CPU time consumed by that function. But, it is defined on the *object* component as the total CPU time consumed by all the method members invoked on that object. According to the different interpretations metrics are planed to be instantiated on components at the instrumentation moment.

Finally, one has to note that the coupling between *SCSM* and *SCMM* is weak. Indeed, from the representation point of view, each class specifying a component of *SCSM* contains as an attribute the list of objects instantiating components of *SCMM*. Consequently, any change performed on *SCMM* lead to a light or no reconsideration of *SCSM*, and *vice versa*. *SCPM* has been extended in [14] such that it deals with distributed software. The next section highlights how the profiling model is instantiated to provide performance data.

4.2.2 A Case Study: Instrumenting the ObjectStore Query Package

To illustrate the implementation of our byte-code profiling model, we consider the ObjectStore Query Package byte-code. The ObjectStore Query Package is a set of *.class* files. Our tool allows the user to instrument the package so that it measures the processing time of his/her queries. To do that we included in the tool the *mocha* [6] freeware decompiler that transforms the *Query.class* file into *Query.java* file. The tool analyzes that file using Javacc [17] and generates an XML-based representation of its structure.

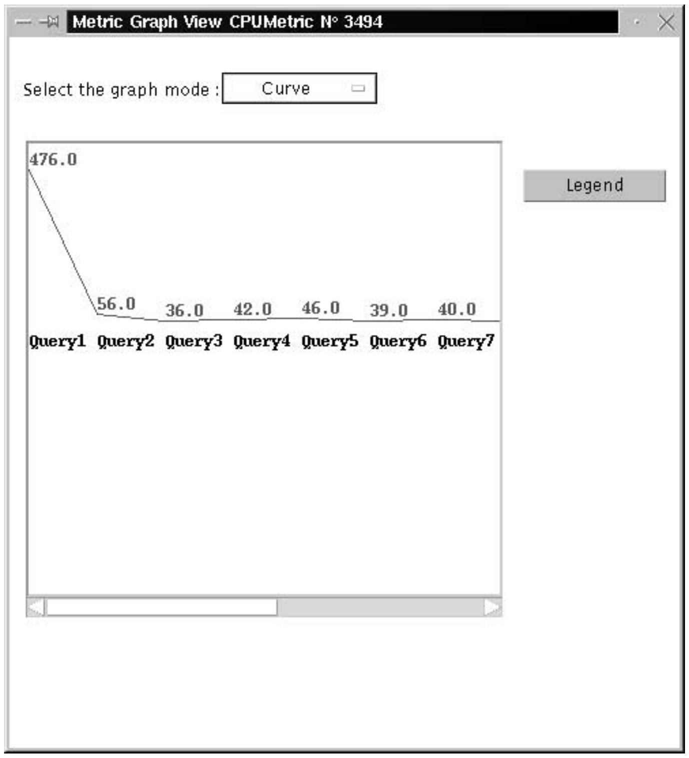


Figure 7: Curve style-based visualization of the query processing times

The XML components designate the different code components and the relationships between them (Figure 3). The structure is then stored into persistent ObjectStore objects as a multi-graph as illustrated in Figure 6. The user can thus select his/her components and metrics and trigger the instrumentation operation that will be done automatically. For example, in Figure 6, the *Select* method code component and the *CPU* metric are selected by the user. The left window contains the instrumented code automatically generated by our tool.

At execution of the instrumented code the query processing times are stored into a text file, namely *MetricResults.txt*. Thus, they can be viewed by our integrated visualization tool. Figure 7 shows the graph visualization of the collected data associated with Query method execution. Various visualization techniques can be used to display metric values (horizontal/vertical histograms, curves).

5 Concluding Remarks

We have proposed and implemented a model for distributed multi-language software code analysis. This allows to represent the software components and their relationships, in an XML representation. The XML representation is generated by using a multi-language parser, which analyze source code and byte-code. The XML representation is stored in an ObjectStore repository, as a multi-graph. The multi-graph provides an uniform way to manipulate the programs. We have considered two kinds of program manipulations : the software change impact analysis and the instrumentation-based profiling.

References

[1] R.S. Arnold and S.A. Bohner. Impact Analysis - Toward a Framework for Comparison. *Proc. of IEEE-ICSM'93*, pages 292–301, 1993.

[2] S.A. Bohner and R.S. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, ISBN 0-8186-7384-2, 1996.

[3] H. Burkhart and R. Millen. Performance Measurement Tools in a Multiprocessor Environment. *IEEE Transactions on Computers*, 38(5):725–737, May 1989.

[4] L. Deruelle, M. Bouneffa, J.C. Nicolas, and G. Goncalves. Local and Federated Database Schemas Evolution: An Impact Propagation Model. *Proc. of DEXA'99, Springer Verlag LNCS 1677*, pages 902–911, Aug. 30–Sep. 03 1999.

[5] Object Design. *Bookshelf for ObjectStore PSE Pro Release 3.0 for Java.* http://www.objectdesign.com, 1998.

[6] D. Dyer. Java decompilers compared. Technical report, http://www.javaworld.com/javaworld/jw-07-1997/jw-07-decompilers.html, July 1997.

[7] B.P. Miller et al. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11), Nov 1995.

[8] D.A. Reed et al. Scalable Performance Analysis: The Pablo Performance Analysis Environment. *In Scalable Parallel Librairies Conference, IEEE Computer Society*, 1993.

[9] A.J. Goldberg and J.L.Hennessy. Performance Debugging Shared Memory Multiprocessor Programs with MTOOL. *In Proc. of Supercomputing'91, Albuquerque, NM*, pages 481–490, Nov. 18–22 1991.

[10] J. Han. Supporting Impact Analysis and Change Propagation in Software Engineering Environments. Technical report, Peninsula School of Computing and Information Technology, Oct. 1996.

[11] F. Lange, R. Kroeger, and M. Gergeleit. JEWEL: Design and Implementation of a Distributed Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):657–671, Nov 1992.

[12] T. Lehr, D. Black, Z. Segall, and D. Vrsalovic. MKM: Mach Kernel Monitor Description, examples and measurements. Technical Report CMU-CS-89-131, Carnegie-Mellon University, March 1989.

[13] N. Melab, H. Basson, M. Bouneffa, and L. Deruelle. Performance of Object-oriented Code: Profiling and Instrumentation. *Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM'99)*, Oxford, UK., Aug. 30 - Sep. 3 1999.

[14] N. Melab and L. Deruelle. Toward a model and a platform for profiling of multi-language distributed object software. *Proc. of PDCS'2000, pages 275-280, Las Vegas, Nevada, USA*, Aug. 8–10 2000.

[15] N. Melab, L. Deruelle, M. Bouneffa, and H. Basson. Instrumentation-based Profiling Techniques. *Proc. of ISCA-CATA'2000, New Orleans, Louisiana, USA*, Mar.29–31. 2000.

[16] N. Melab, N. Devesa, M-P. Lecouffe, and B. Toursel. A Periodic Adaptive Strat-

egy for Monitoring Distributed Applications: Description and Analysis. *Proc. of HPCS'97, Winnipeg, Manitoba, Canada*, pages 645–654, July 10-12 1997.

[17] Sun Microsystems. Java Compiler Compiler Documentation. Technical report, http://www.sun.com/suntest/products/JavaCC /DOC/index.html, Jan. 1999.

[18] V. Rajlich. A Model for Change Propagation Based on Graph Rewriting. *Proc. of IEEE-ICSM'97, Bari, Italy*, pages 84–91, Oct. 1–3 1997.