A Change Propagation Model and Platform For Multi-Database Applications

L Deruelle, Mourad Bouneffa, N. Melab, Henri Basson

A Change Propagation Model and Platform For Multi-Database Applications

L. Deruelle, M. Bouneffa, N. Melab, and H. Basson
Laboratoire d'Informatique du Littoral
Maison de la Recherche Blaise Pascal
50, rue Ferdinand Buisson - BP 719
62228, Calais Cedex, France
{deruelle, bouneffa, melab, basson}@lil.univ-littoral.fr

Abstract

In this paper, we propose a formal model and a platform for software change management. The model is based on graphs rewriting, and deal with both multi-language source codes and heterogeneous database schemas. These are represented by software components linked by meaningful relationships. The change impact analysis is done, using a Knowledge-Based System, that includes impact propagation rules preserving the software consistency. This is implemented by an integrated platform including a multi-language parsing tool, and a software change management module.

Keywords : *Source Code Analysis, Software maintenance, Distributed database applications, Change Propagation, Knowledge-Based Systems.*

1. Introduction

Nowadays, observing their information systems, enterprises are confronted with two main problems. The first one concerns the multiplicity of the used languages, database systems, operating systems and software environments. This leads to complex applications based on the cooperation of different subsystems. The second problem concerns the frontiers of the information system. Supply chain management and e-business challenges lead the information systems to deal with suppliers and customers ones. Strictly internal information systems will not be sufficient for such application areas. So, distributed and complex applications strongly emerge. These require complex object modeling, scalability, integration of large-scale databases and programming facilities. For that, distributed system middleware solutions like CORBA [22][12], DCOM [9] and Java RMI may be used, as shown in figure 1.

Regarding such applications, design and implementation seem to be critical tasks. On the other hand, the evolution and the maintenance of these applications is not obvious. Without tools and methodologies to control it, the software change may cause serious damages to the information system.

The quality standard ISO 9126 defines four components for the maintenability : analyzability, testability, stability and changeability [6]. In this work, we focus on the software analyzability and changeability. The challenge is to analyze and to manage the changeability of real size applications, based on multi-language programs, and heterogeneous databases, within a CORBA-based distributed framework. One of the ways to manage software changeability is to assess the change impacts, and to re-establish the system consistency, using a change propagation process.

Many efforts have been addressed to deal with software change management issues, such as identifying the need for change, assessing the impact of the proposed [1][4], developing methods for the change process [3][16][17], managing the versions of the modified software artifacts, collecting change-related data. In comparison with some models [2][20][21], the major advantage of our approach concerns the deal with multi-language program source codes and heterogeneous databases, in both centralized and distributed environments.

The paper is organized as follows : Section 2 presents our model, called Software Components Structural Model (SCSM). It represents the software components and their relationships by means of graphs. Section 3 provides a taxonomy of the considered software changes. Section 4 describes the change propagation process, based on a knowledge-based system. Section 5 explains the global architecture of our platform, that implements the formal models and includes parsing tool for multi-language source code analysis. In section 6, we illustrate the global mechanism

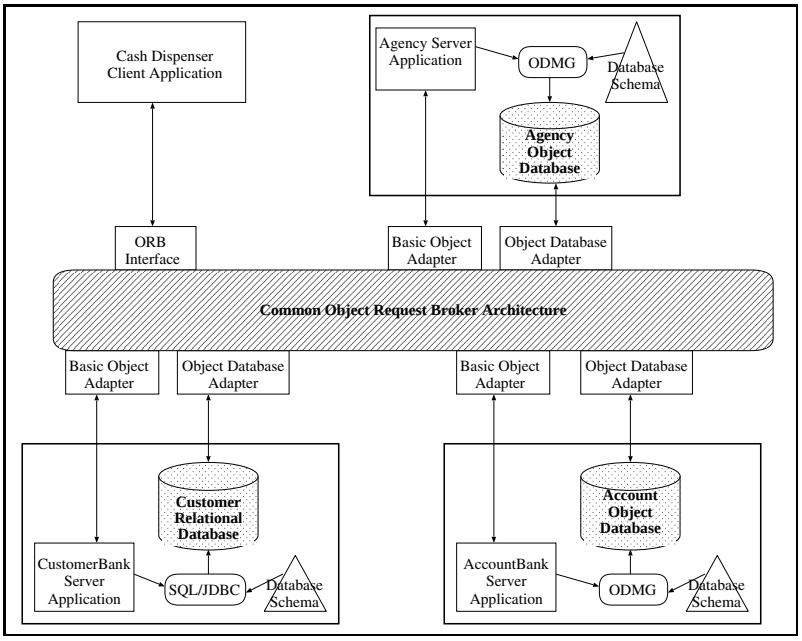

Figure 1. Representation of a Bank Application based on the CORBA middleware

with an example. Finally, section 7 gives some concluding remarks.

2. SCSM : a Formal Model for the Change Propagation

The Software Components Structural Model (SCSM) is the core element of our approach [18]. It represents the various and may be heterogeneous software components and their relationships. Such components may be *classes*, *methods*, *relational tables*, *persistent objects*, etc. Relationships may be *inheritance function or method calls*, etc. V. Rajlich [21], D.Kung [15], and R. Keller [6] have formalized programs by graphs, in which the nodes are the software components, and the edges are the dependency relationships between them. We adopt a similar approach with the following refinements :

- The nodes are typed. This allows a partition of the nodes and a dynamic construction of derived partitions during the change impact analysis.
- The edges are induced by the various relationships and every relationship type may define a partial software graph.
- The software graphs are constrained. The constraints represent the invariant properties of the software. Such properties define the software consistency and must always be preserved.

The main advantage of these refinements is to make the change impact analysis more flexible. For instance, it will be possible to analyze the impact of deleting a class method on only the inheritance and method call relationships. The constraints may represent language dependent properties or more specialized ones like enterprise specific programming standards. So, the change analysis may be customized for the specific requirements.

Figure 2 shows a set of persistent Java classes used for a bank agency management. The persistence is achieved by inheriting the class *Ipersistent* that is provided by the Object Oriented Database Management System ObjectStore. The corresponding software components graph is $G =< V,\ E,\ Constr >$ defined as follows :

- the set of nodes or vertices V may be partitioned into two sets : *classes* and *methods*.
- the set of edges $E \subset VXV$ is induced by the inheritance, composition, call and use relationships.
- the constraints contains those related to the construction of inheritance graph, functions or methods call, etc. For instance, the inheritance graph must be a connected tree.

Let us consider the partial graph induced by the inheritance relationship.
This graph is $Gh =< Cl,\ Eh,\ Constr >$.

- *Cl* = {*Ipersistent, Paccount, Pcustomer,PAgency*}

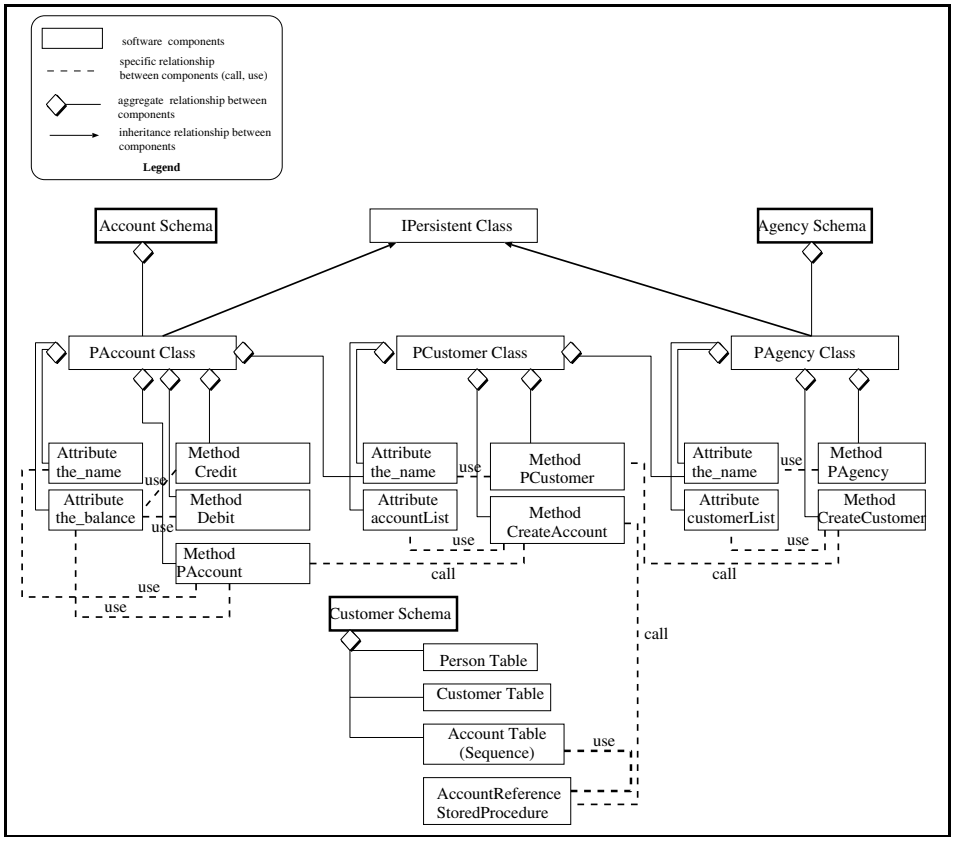

Figure 2. UML-based representation of the bank application

- $E_h = \{< Paccount,\ Ipersistent >,$
  $< PCustomer,\ Ipersistent >,$
  $< PAgency,\ Ipersistent >\}.$
- Let E_h^* be the transitive closure of E_h.
- The set of constraints includes the following expressions meaning that the Java inheritance graph must be a connected tree :
  - $\forall\ n\ \in\ Cl, n\ =\ Ipersistent\ or\ \exists\ n_i\ \in\ Cl$ such that $< n,\ ni >\ \in\ Eh$
  - $\forall\ n_1,\ n_2 \in\ Cl,\ < n_1,\ n_2 >\ \in\ E_h^*\ and$ $< n_2,\ n_1 >\ \in E_h^*\ \longrightarrow\ n1 = n2.$

In the rest of the paper we especially focus the software components related to source codes (imperative and object-oriented) and database schemas (both relational and object-oriented databases).

3. Software Change Identification

The software system is in constant evolution, that leads to changes performed by several persons. One of the major difficulties in software evolution is to automatically identify the changes and their impacts.

In this section, we first classify and formalize the different change types.

The tables 3 and 4 summarize a change taxonomy of source Codes and database schemas (both object-oriented and relational ones). This is an extended classification of the proposed one in [15]. Our taxonomy is based on three granularity levels, defined as follows:1) the coarse granularity level that provides a global view of the application. This level includes components like files, libraries, packages, database schemas. 2) the medium granularity level representing components like classes, functions, global variables, relational tables, stored procedures, etc. 3) the fine granularity level that describes statements, queries, control structures, etc. For every component type, we define two main change operations that rae *add* and *delete*. Such operations may then be combined to define more complex ones. For instance, modify the body of a method may be composed of deleting the old body and adding the new one. Let us explain some change operation and especially those concerning database schemas :

- *Name Change*. A name (persistent root) is an entry

Software Components			Changes
	Sub-components	Relationships	
File, Module, Package, Library			add/delete a file, module, package, library
		importation	add/delete an importation link
Type	Class		add/delete a class declaration
		instantiation	add/delete an object instantiation link
	Interface	inheritance	add/delete an inheritance link add/delete/redefine inheritance restriction
		implementation	add/delete an implementation link
	Structure, Union, Enumerate		add/delete a structure/ union/ enumerate declaration add/delete a field declaration
	Atomic		add/redefine an atomic type
Function/ Method	Prototype		add/delete the prototype declaration add/delete a parameter declaration add/delete the return type add/delete/redefine the access attribute
	Body		add/delete the body
		override	add/delete an override link of a method
		call	add/delete a call to the function/method
Variable	Object, Member, Parameter Local/Global Variable		add/delete the variable declaration add/delete the type of the variable
		use	add/delete a variable use link with a statement
Statement			add/delete a statement add/delete a symbol inside the statement

Figure 3. Source code changes identification

point to store or retrieve a set of persistent objects in an object oriented database [5]. The name deletion causes the deletion of all persistent objects, that are not stored in another name, and disturbs the queries, that perform operations on it.

- *Stored Procedure/Method Change*. A stored procedure defined in a relational database schema, or a method defined in a persistent class can be changed, in the same way than the function and method change.

- *Transaction Change*. Database applications that uses persistent data are organized in transactions. The Object Model, defined by Object Database Management Group, supports a nested transaction model. The transaction can be changed by adding/deleting it, by redefining the level of a nested transaction or by modifying the transaction operations (commit, abort, checkpoint, abort-to-top-level)[5].

- *Query Change*. A query is a statement that manipulates simple or complex data, stored in a database. The query can be changed by updating the entry point (table, name, etc.), the collection structure returned by the query, or the data selected clause.

3.1. Software Change Formalization

Let $G_{sc} =< V_{sc}, E_{sc}, Constr >$ be the graph for software components sc, and $G'_{sc} =< V'_{sc}, E'_{sc}, Constr >$ be the graph for the modified version sc' of the software component sc. We consider two ways to modify the version of the software component sc:

- *the structural modification* is expressed by $V'_{sc} = Change(ModificationType, V_{sc})$,

- the *relationships modification* is expressed by $E'_{sc} = Change(ModificationType, E_{sc})$.

- the set of constraints $Constr$ must always be satisfied.

Formally, a change may then be expressed by a graph morphism f defined by the expression $G'_{sc} =< V'_{sc}, E'_{sc}, Constr >= f(< V_{sc}, E_{sc}, Constr >)$. The graph morphism f is composed of two functions denoted f_{node} and f_{edge} such that $f_{node}(V_{sc}) = V'_{sc}$ and $f_{edge}(E_{sc}) = E'_{sc}$. Preserving the set of constraints is not obvious. This may lead to reapply other morhphisms as side effects or impacts of the original one. The recursive application of morphisms denotes the impact propagation process. For instance, let $Change(deleteclass, C)$ be the

Database Software Components			Changes
	Sub-components	Relationships	
Schema			add/delete a schema
		importation	add/delete an imported schema
Persistent Class			add/delete a peristent class declaration
		implementation	add/delete an implementation link
		inheritance	add/delete a persistent superclass link add/delete/redefine inheritance restriction
		instanciation	add/delete a persistent object instantiation link
Table			add/delete a table
		cardinality	add/delete a cardinality link between tables
Name (persistent root)			add/delete a name add/delete the type of the name
Stored Procedure/ Method	Prototype		add/delete the prototype declaration add/delete a parameter declaration add/delete the return type add/delete/redefine the access attribute
	Body		add/delete the body
		override	add/delete an override link of a method
		call	add/delete a call to the function/method
Variable	Persistent Object, Member, Parameter		add/delete the variable declaration add/delete the type of the variable
		use	add/delete a variable use link with a statement
Transaction			add/delete a transaction redefine the level (nested transaction) add/delete a transaction operation
Query			add/delete a query add/delete a clause inside the query

Figure 4. Relational and Object Oriented Database changes identification

class deletion operation. The corresponding morphism is defined by :

- $f_{node}(V_{sc}) = V'_{sc}$=$V_{sc} - C$.
- $f_{edge}(E_{sc}) = E'_{sc}$=$E_{sc}$-$outedges(C) \cup inedges(C)$ ($outedges(C)$ and $inedges(C)$ are two functions returning the edges coming from C or entering to C).

Such a morphism causes a violation of inheritance graph constraints. Assume that the class C is a Java class, so the inheritance graph will become a not connected one. In the following section we show how we propagate the impact to fire the recursive execution of the other morphisms.

4. 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 system consistency, by making a set of estimated consequent changes [13]. This process would involve advising 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* [21].

The *change-and-fix* algorithm allows to perform a change on a selected graph node $a \in ent(P)$, the set of the entities or components of a program P, and computes the modified program, referred to as P'. This permits to mark the nodes and dependency relationships that are inconsistent in the program. The process iterates until the set of marked nodes becomes empty. In this algorithm the expressions $P(a)$ and $P'(a)$ represent the neighborhood (incoming and outgoing nodes) of a component or entity a.

The change-and-fix algorithm is :

Given a consistent program P
Select $a \in ent(P)$
Change(a)
$P = (P - P(a)) \cup P'(a)$;
do {
    Select a $\in$ (mark(P) ;
    Change(a) ;
    $P = (P - P(a)) \cup P'(a)$;
}
while $(mark(P) \neq \emptyset)$;

We can identify two problems with the previous algorithm :

- the entities of a program 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, this leads to mark all the neighbors of this node or no one of them. This is a consequence of considering all the relationship types as a unique one called dependency relationship. However, if we take into account the different semantics of relationships we can refine the impact propagation to the direct neighbors by marking those really affected by the change.

We propose a change propagation algorithm based on an expert system that provides solutions to the quoted problems.

4.1. Solution based on an Expert System

We propose a hybrid implementation of the *change-and-fix* process with combining a graph rewriting algorithm and a knowledge-based system. So, the impact of changing a node is propagated by a change and fix algorithm depending on both the neighborhoods of this node and the constraints that are represented by a set of rules. The knowledge-based system is built using the RETE algorithm [10]. This algorithm is widely recognized as by far the most efficient algorithm for the implementation of rule-based systems. The RETE algorithm efficiency is asymptotically independent of the number of rules. The hybrid implementation provides a change propagation, provided by the *change-and-fix* algorithm and deals with more explicit knowledge represented by rules and facts. As an example of such rules, the following marks a modified class and propagates the change to its related components, such as the methods :

```
(defrule ClassImpactRule
  (Delete (OBJECT ?Class))
  =>
  (set ?Class marked TRUE )

  (bind ?Methods (get ?Class methodList))
  (while (call ?Methods hasMoreElements)
   (definstance Method
    (bind ?Method (call ?Methods nextElement)
  )))
  ...
)
```

This rule means when a class is deleted, the system marks it and explores its related components like methods and the classes that effectively call these methods.

The change-expert-system algorithm is :

given a graph of a consistent
program $G =< V, E >$
select $a \in E$
/ change the node or the edge a */*
$a' = Change(ModificationType, a)$;
/ expert system features */*
insert fact (ModificationType, a');
/ compute the modified graph G' by using*
/ the corresponding morphism of*
/ ModificationType*/*
$G'_{sc} = (V'_{sc}, E'_{sc})$
do {
  do {
    / Rete inference process */*
    Select a rule r in fired rules set,
    using RETE;
    / insert new facts */*
    Trigger $r \cdot actionSet$;
  } while (fired rule set $\neq \emptyset$)
  / change-and-fix features */*
  select a $\in$ (mark(G) ;
  $a' = Change(ModificationType, a)$;
  insert fact (ModificationType, a');
  $G'_{sc} = (V'_{sc}, E'_{sc})$
} while $(mark(G) \neq \emptyset)$;

The change-expert-system algorithm deals with the change-and-fix problems, by inserting only new facts in the factual database. This avoids to revisit the software components. The system inference provided by the RETE algorithm is based on rules management and pattern matching, dealing with the infinite graph loops. In contrast with the strictly procedural change-and-fix algorithm [21], only the affected components are visited. The set of active rules may be customized by the user. For instance, it is possible to consider only one relationship type like inheritance without considering the others like call one. This leads to an incremental impact analysis process.

The formal model and the knowledge-based system are implemented by a distributed platform, providing change propagation. Doing this, we deal with the distributed heterogeneous software, including database systems.

5. IFSEM : Integrated Framework for Software Evolution and Maintenance

In complex distributed applications, the program source codes and the database schemas are distributed over many computers and networks. For the change impact analysis, the software components are distributed, in the same way.

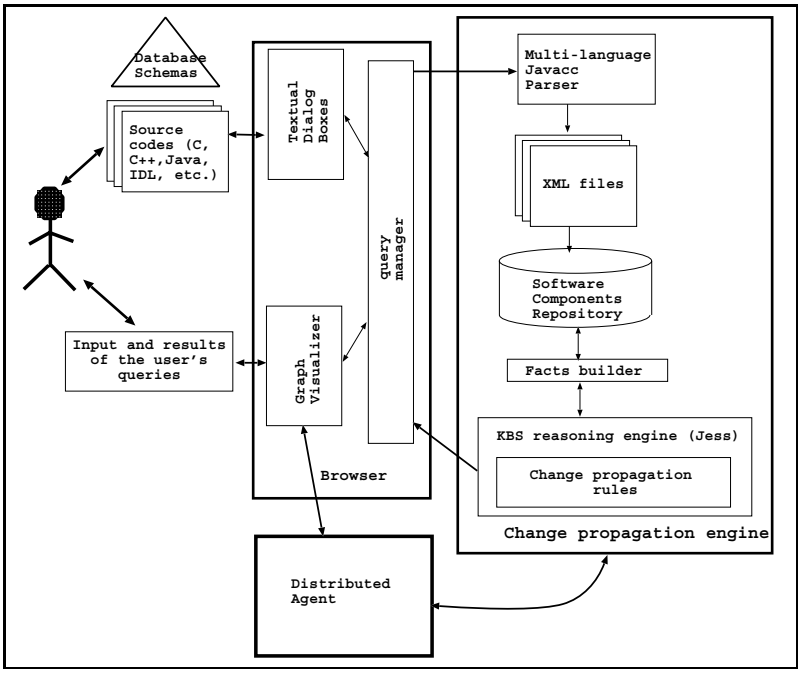

Figure 5. The Distributed Framework Architecture for Change Impact Propagation

In order to take into account the distributed relationships between software components, especially database schemas, we need to distribute our graphs, following the distributed architecture of the software. The distributed graphs are connected by the way of distributed agents.

The model have been implemented by a framework, called *Integrated Framework for Software Evolution and Maintenance*. This provides change impact propagation for both centralized and distributed software. The figure 5 shows the framework architecture, for distributed change impact propagation. It provides incremental and multi-language source code files and distributed database schema files parsing to extract structural information, using *Javacc parser* [19]. The structural information represents the software components and their relationships, which are described by XML files (eXtensible Markup Language). The XML files are used to construct the software multigraph representation, which is stored in a distributed component repository. The component repository is implemented with *ObjectStore Database* [8] , allowing to store complex object graphs. The *query manager* allows the programmer to simulate its change. The change impact propagation is computed by the *Change Propagation Engine*, using the expert system, implemented using *Jess*[11] [7]. This propagates the change impact locally, using propagation rules, and marks the graph nodes, with the *affected* label. In a distributed environment, the Expert System propagates the change impact, using the *distributed agent*. This broadcasts the affected components set to other distributed agents. These insert the new facts in their local *Change Propagation Engine*, in order to perform the change impact analysis, in the distributed level. The distributed agents broadcast the result of the change impact analysis, in order to inform the programmer of the change effects, in the distributed level.

We present, here after, an example of a change impact analysis, performed on a CORBA-based bank application.

6. Application : Evolution of a CORBA-based bank Multi-Databases

Figure 1 shows a global view of the bank application. This is composed of three distributed and heterogeneous database schemas :

- *The Agency schema*, is an *ObjectStore* ODMG-compliant database, that stores all transactions related to the customers and their accounts. The Agency database system is running on a Pentium II 350MHz with a Linux Operating System.

- *The Customer schema*, is an *Oracle 8i* relational-object database, that stores the bank customers, related to an agency. The Customer schema defines tables and stored procedures to store information related to consumers and bank account references. The Oracle database is running on a Pentium II 450MHz with the Linux operating system.

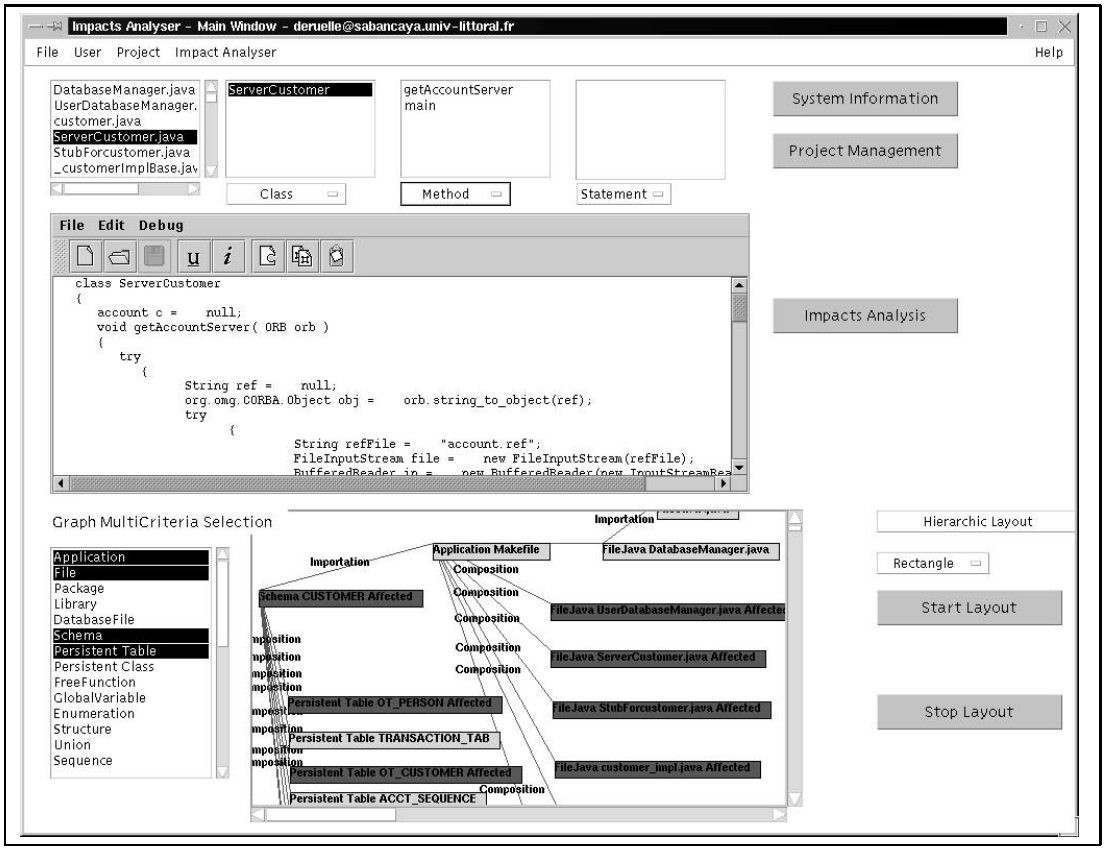

Figure 6. Result of the change impact propagation at the distributed level

- *The Accounts schema*, is an *ObjectStore* ODMG-compliant database, that stores all bank accounts related to an agency. This database is running on a Silicon Graphics workstation with IRIX5.3 operating system.

These database schemas are connected by the Object Request Broker, called ORBACUS[14].

The *Agency Server Application* manages agencies, the customers and the bank accounts, in a distributed manner. The *Agency Application* provides customers related operations, using the *Customer Server Application*, like customers registration, and accounts operations like creation, deletion, debit and credit, using the *Account Server Application*. The *Cash Dispenser* allows the customer to query the *Agency Server Application* to perform debit or credit operations.

In this example, we propose to perform a change, related to the Account schema. The considered change is the deletion of the schema *Account* (figure 2). The schema *Account* is implemented with a persistent class *PAccount*, using the ObjectStore database system, which is used by the *Account Server Application*. The Change Propagation Engine computes the change impact propagation locally. So, the local impact propagation engine marks the class $PAcount$ and its methods as inconsistent ones. Considering the *Connection Relationship* between the *Account Server Application* and the Account Object Database, the change propagation engine propagates the change to the software components belonged to the *Account Server Application*. Inserting the following fact, which fires change propagation rules performs the change propagation,:

```
/* insert the fact representing */
/* the account schema deletion  */
(Delete Account)

(defrule SchemaImpactRule
 (Delete (OBJECT ?Schema))
=>
 /* the rule marks the schema component */
 /* as affected by the change, */
 (set ?Schema affected TRUE )

 /* and propagates the impact, */
 /* using the Connection relationship. */
 /* The rule retrieves all components, */
```

```
/* connected to the deleted schema component, */
(bind ?Connections (get ?Schema ConnectedTo))
(while (call ?Connections hasMoreElements)

   /* and inserts the Connection fact, */
   /* to propagates the change impact. */
   (definstance Connection (bind ?Connect
      (call ?Connections nextElement) )
   )
)
...)
```

The *SchemaImpactRule* propagates the change impact, using the connection relationship, and marks affected software components, belonged to the *Account Schema* and to the *Account Server Application*, such as the $PAccount$ method.

The *Customer Server Application* is connected to the *Account Server Application* to retrieve account information's related to the bank customers. Considering the "affected" software components belonged to the *Account Server Application*, the *agent* propagates the change impact to the distributed agents, using the CORBA middleware. The distributed agents insert the facts, representing the software components, in their change propagation engine.

Regarding the *PCustomer* class and the fact *(Delete PAccount)*, the call relationship between the methods $PAccount$ (PAccount class) and $CreateAccount$ (PCustomer class) generates new facts, which are used to fire change propagation rules and to mark the $CreateAccount$ method. In the same way, the call relationship between the $CreateAccount$ method and the *AccountReference* stored procedure, which is defined in the Oracle Customer Schema, leads to propagate the change impact in the *Customer Schema* and in the *Customer Server Application*.

The figure 6 shows the change propagation on the Customer Relational database schema, at the distributed level. In this figure the marked nodes are labeled with the symbol "affected" and are colored with the red color.

7. Conclusion

We have proposed and implemented an approach for the change impact analysis of the distributed database applications. This has been applied to propagate the database schema change impact to the application programs that may be a part of a distributed software. We consider the software as distributed and heterogeneous databases and multi-language source codes.

The multi-database schemas and program source codes are represented by graphs, that implement our proposed Software Components Structural Model (SCSM). This leads to propagate the change impact by navigating through the paths of graphs. The SCSM takes into account all kind of software components even if they represent large ones like files and database schemas or more fine ones like statements, queries and individual symbols. This makes the analysis more exhaustive. We are refining the relationships between the distributed components within the framework of databases interoperability, provided by the standard services of the Object Management Group and the Object Database Management Group (CORBA).

The prototype, called Integrated Framework for Software Evolution and Maintenance that implements our approach is based on a knowledge based system (KBS), that makes it more flexible. We deal with the distributed change impact propagation, using distributed agents, that provides the graphs interconnection and change impact broadcasting to other distributed agents.

Our model and the platform are being extended along three areas,

- Firstly, the KBS is extended to provide a semi-automatic programs restructuring tool.
- Secondly, we are experimenting our platform to deal with the evolution and maintenance of web based applications.
- Finally, we are planning to integrate our platform with architecture definition languages. So, we expect to propagate a change at architectural level to the implementation one and vice-versa.

References

[1] R. Arnald and S. Bohner. Impact analysis - towards a framework for comparison. *Proc. of the International Conference on Software Maintenance (ICSM'93), Montreal, Canada*, pages 292–301, Sep. 1993.

[2] D. Atkinson and W. Griswold. The Design of Whole-Program Analysis Tools. In I. C. Society, editor, *The proc. of the 18th International Conference on Software Engineering, Berlin*, March 1999.

[3] D. Avrilionis, P.-Y. Cuin, and C. Fernstrm. Opsis: A view mechanism for software processes which supports their evolution and reuse.

[4] S. Barros. *Analyse a priori des consquences de la modification de systmes logiciels : de la thorie la pratique*. PhD thesis, Universit Paul Sabatier Toulouse, 1997.

[5] R. G. G. Cattel. *ODMG-93 Object-Oriented Databases Standard*. International Thomson Publishing, 1995.

[6] M. Chaumun, H. Kabaili, R. Keller, and F. Lustman. A Change Impact Model for Changeability Assessment in Object Oriented Software Systems. *Proc. of the Third IEEE Euromicro Working Conference on Software Maintenance and Reengineering, Amsterdam, The Netherlands*, pages 130–138, Mar. 1999.

[7] L. Deruelle, M. Bouneffa, J. Nicolas, and G. Goncalves. Local and Federated Database Schemas Evolution: An Impact

propagation Model. In L. N. in Computer Science, editor, *The proc. of the Database and Expert Systems Applications*, 1999.

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

[9] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1999.

[10] C. L. Forgy. Rete: A fast algorithm for the many pattern/ many object pattern match problem. *Artificial Intelligence 19*, Addison-Wesley:17–37, 1982.

[11] E. J. Friedman-Hill. *Jess 4.3 User's Manual*. Sandia National Laboratories, December 1998.

[12] J. Geib, C. Gransart, and P. Merle. *Corba des Concepts la Pratique*. InterEditions, 1997.

[13] J. Han. Supporting impact analysis and change propagation in software engineering environments. Technical report, Peninsula School of Computing and Information Technology, oct. 1996.

[14] http://www.ooc.com/. *ORBACUS : user guide*, December 1999.

[15] D. Kung, J. Gao, P. Hsia, and F. Wen. Change Impact Identification in Object Oriented Software Maintenance. *Proc. of the International Conference on Software Maintenance (ICSM'94), Victoria, B.C., Canada*, pages 202–214, sept 1994.

[16] M. Lehman. Process models, process programs, programming support. In *Proceedings of the Ninth International Conference On Software Engineering*, March 1987.

[17] J. Lonchamp. Supporting Social Interaction Activities of Software Processe s. In J.-C. Derniame, editor, *Proceedings Second European Workshop Software Process Tech nology*, pages 34–54, Trondheim (Norway), September 1992. Springer Verlag. Lecture Notes in Computer Science, 635.

[18] 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.

[19] S. Microsystems. *The Java Compiler Compiler Documentation*. http://www.sun.com/suntest/products/JavaCC/, January 1999.

[20] G. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, July, 1996.

[21] 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.

[22] R. Zahavi and T. Mowbray. *The Essential CORBA-Systems Integration Using Distributed Objects*. John Wiley and Sons, Inc, 1996.