A CHANGE PROPAGATION PROCESS FOR DISTRIBUTED SOFTWARE ARCHITECTURE

Mohamed Oussama Hassan, Laurent Deruelle, Henri Basson, Adeel Ahmad
Université Lille Nord de France
Laboratoire d'Informatique, Signal et Image de la Côte d'Opale
50 rue Ferdinand Buisson BP 719, 62228 Calais Cedex, France
{*hassan,deruelle,basson,ahmad*}*@lisic.univ-littoral.fr*

Keywords: Distributed Software Architecture analysis, Change Propagation, Knowledge-Based System.

Abstract: In the context of software architecture evolution, understanding the impacts of a change to be applied on a distributed software architecture is necessary for various activities related to maintenance and change management. In this paper, we propose formal models and a platform based on eclipse plugins for modeling and analysis of the software architecture description and their related source codes. The proposed models aim at the construction of graph representation based on the architecture description and the software source codes. The graph implementation is mapped with facts of a distributed knowledge-based system, which performs change propagation rules to evaluate the impact of a change performed on distributed components.

1 INTRODUCTION

Nowadays practices of software engineering are often asked to respond to development or evolution of distributed applications on heterogeneous platforms with less delay, lower cost, and better quality. The applications are continuously growing in size and complexity making the change more difficult to control and to manage. Moreover, it is largely admitted that the quality of large applications can be improved using formalized architectural models at the earlier phases of requirements specifications and design. Therefore, over the past decade software architecture has received increasing attention as an important subfield of software engineering aiming to face the growing size and complexity of software (Taylor et al., 2009).

It is inevitable that a software undergoes some changes in its lifetime. A change introduced to one component of an application has often effects on several others parts. If this process is uncontrolled, changes may have unexpected side effects or complex implications on the behavior of the whole system. The cost to fix an error or an incoherence, resulting from changes during requirements, or early design phases, is largely lower than the cost of correcting the same error found during system testing or in production (Clements et al., 2002). The change impact analysis refers to track the effects of a change by providing visibility of the potential effects of the proposed change before it is implemented. It greatly helps maintainers to determine appropriate actions to take with respect to change in decisions, schedule plans, cost and resource estimations.

2 OBJECTIVES

In this paper, we propose a model, called Architecture Software Components Model (ASCM), to represent and to unify the major concepts defined in a large number of existing architecture description languages (ADL). The ASCM is coupled to another model called Source Code Structural Model (SCSM) (Deruelle et al., 2001; Deruelle et al., 2007) in order to represent relationships between components from the architecture level and those of the source code level. The two models represent the architecture and the source codes as graphs, in which the nodes are components and the edges are relationships. The graphs are distributed according to the location of the architecture specification and the relevant source codes. The main purpose of our models is to provide a basis to track the change propagation process using adequate graphs on which we define formal propagation rules.

The rest of the paper is organized as follows : Section 3 presents related works for distributed software architecture evolution. Section 4 describes our formal model ASCM to represent the common concepts defined in the major ADLs. Section 5 proposes a typology of formalized change operations applied on software architecture and permitting the change impact analysis. Section 6 defines a formal process to deal with the change propagation process for distributed software. Section 7 presents our integrated platform that implements the formal models and a knowledge-based system to support the change propagation process. Section 8 gives a scenario of a change impact analysis performed on a distributed software architecture. Finally, section 9 provides some conclusions and perspectives.

3 DISTRIBUTED SOFTWARE ARCHITECTURE EVOLUTION : RELATED WORKS

Software architecture has emerged as an area of intense research over the last decade. Many approaches have been proposed to deal with architectural specification and analysis (Medvidovic and Taylor, 2000; Bass et al., 1998; Clements and Shaw, 2009). In these approaches, a large number of ADLs have been developed to represent different aspects of distributed software architectures based on the middlewares. The ADLs do not provide features to deal with the evolution management of architecture description in a distributed environment.
Many works have been done for studying the evolution mechanism in the ADLs. We note that the specification of the evolution is realized by the ADL that serves for the architecture description. Therefore, the evolution management is included in the architectural specification, and so it will be difficult to be distinguished. Moreover, it is almost impossible to identify all possible evolutions operations that may occur when specifying architecture.
It is hard to deal with the non planned evolution, considering the difficulty to study and analyze automatically the changes impact without incoherence. The number of propositions dealing with architecture change impact analysis is very restricted.

4 DISTRIBUTED SOFTWARE ARCHITECTURE MODELING : ASCM

Many resarch works have been done to provide architectural modeling (Garlan et al., 1997), in which the following common high-level :

1. *the Components* are units of computation or a data stores. For example, a component can be a Thread, a Procedure or a whole application. An instance of a component may be distributed on many sites and may interoperate with other distributed components.
2. *the Connectors* are architectural building blocks used to represent the interactions between local and distributed components. For example, a $Thread$ deployed on one site can be connected to another $Thread$ deployed on another site to send messages. The second component may treat messages using local components.
3. *the Configurations* are connected graphs of local and distributed components and connectors that describe the structure of the distributed architecture.

Although these concepts define a high-level common structure for the distributed software architecture, we need to refine them in a more accurate model to describe more precisely the structure of a distributed architecture.

The model ASCM leads to represent the distributed software architecture, based on architecture specifications described using different ADLs. Our work consists to represent in a model the common high-level concepts defined in the major ADLs. This allows us to analyze various architectural description documents, which specify distributed software architectures, in order to extract the elements and to represent them using ASCM.

By comparing in more detail the definition of the ADLs, we can refine these elements to provide a more precise model. Let us describe the ASCM model presented in figure 1 :

- The *interface* defines the services that other components wants to invoke. An *interface* may define *ports* used by a *connection*. A *port* can be used by several connectors. In the ADLs, an interface is mandatory to describe a distributed component in the architecture.
- The *implementation* reflects the source codes of the services defined in the *interface*. For example, the *implementation* can describe how a component calls sub-programs to realize the services.

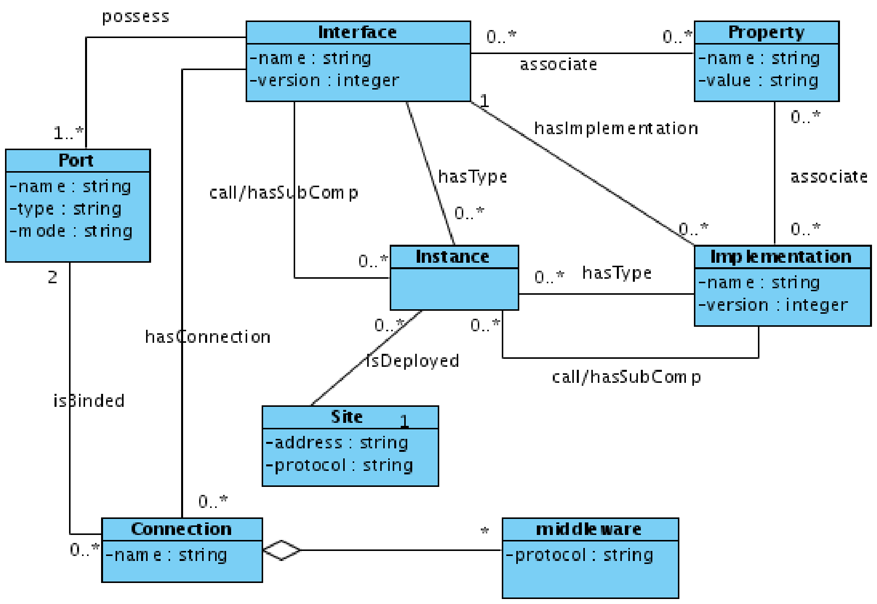


Figure 1: ASCM: UML-based representation of Architecture Description Languages

- the *connection* provides a link mechanism between distributed components to exchange information. The link is usually managed using a *middleware* that allows a component deployed on one system to access programs and data on another one.
- the *middleware* is a set of services that allows interactions between multiple components running on many sites. It helps to solve heterogeneity, interoperability and distributed computed problems.
- The *instance* represents a sub-component belonging to a component. The sub-component is instantiated from the interface or the implementation of another component.
- The *site* is the element on which the instances of a component will be deployed on. It defines the hardware and software requirements for the instances running.
- The *properties* can specify constraints on the interface or the implementation of a component. The constraints may be functional or not, such as related source code of a component, performance, reliability and availability.

We coupled ASCM with SCSM, which is fully described in (Deruelle et al., 2001; Deruelle et al., 2007) by formalizing a generic relationship that may exist between their respective elements. The relationship will be used for propagating the impact of a change done on the architectural description to its corresponding source code and reciprocally. The models SCSM and ASCM are coupled by a projection relationship. When a *projection relationship* exists, the software modeling is more precise and the change impact analysis is performed on the architectural level and the source codes one.

Hereafter, we propose a typology of formalized change operations applied on the distributed architecture, which are represented by ASCM model instances.

5 TYPOLOGY OF CHANGE OPERATIONS

A typology of change operations is presented in the table 1, using the assertion formalism. Assertions are widely used in the software community to formalize the programs behavior (Mens, 2001). For each operation, we specify a pre-condition to be checked in order to allow the execution of the operation. The post-condition indicates the operation results. The invariants represent the propositions to be verified before and after the operation execution.
Hereafter, we describe a change operation to illustrate the assertion formalism and the impact on the distributed architecture and on the source codes.

Let us to consider the operation $del\_comp\_imp(c, cimp)$, which consists to delete the

Table 1: *Typology of change operations applied on Software Architecture*

Name	**Operation**	**Pre-condition**	**Post-condition**	**Invariant**
Component interface adding	$add\_int\_comp(c,t)$	$-c$ $+t$	$+c$	$-edge(*,c)$
Component implementation adding	$add\_imp\_comp(cimp,c)$	$-cimp$	$+cimp$ $+edge(*,c,cimp)$	$+c$
Port adding	$add\_port(a,p)$	$-p$	$+p$ $+type(p,port)$ $+edge(hasPort*,a,p)$	$+a$
Subcomponent adding	$add\_sub\_comp(a,s,t)$	$-s \vee$ $(+s \wedge -edge(e,a,s))$	$+s$ $+edge(e,a,s)$ $+type(e,hasSubComp)$	$+a$ $+t$
Component interface deletion	$del\_comp\_int(a)$	$+a$	$-a$ $-impl(*,a)$ $-edge(*,a)$	$-subComp(*,a)$ $(-subComp(*,ai),$ $+impl(ai,a))$
Component implementation deletion	$del\_comp\_imp(c,cimp)$	$+cimp$	$-cimp$ $-edge(*,cimp)$	$+c$ $-subComp(*,cimp)$ $-source(hasSubComp*,$ $cimp)$
Connector deletion	$del\_conn(cn,c,a,b)$	$+cn$ $+edge(hasConn*,c,r)$ $+edge(usePortSrc*,cn,a)$ $+edge(usePortDst*,cn,b)$	$-cn$ $-edge(*,cn)$	$+a$ $+b$ $+c$
Port deletion	$del\_port(a,p)$	$+p$ $+type(p,port)$ $+edge(e,a,p)$	$-p$ $-edge(*,p)$	$+a$ $-dest(usePortSrc*,p)$ $-dest(usePortDst*,p)$
Subcomponent deletion	$del\_subcomp(a,s)$	$+s$ $+edge(hasSubComp*,a,s)$	$-s$ $-edge(*,s)$	$+a$ $-connSubComp(*,s)$

Table 2: *Assertions list for the graph marking*

Assertion	**Signification**
$mark(el)$	$\forall el \in E \vee el \in N,\ state(el) = affected$
$mark(v,op)$	$\forall e \in E, v_1 \in N : ((+edge(e,v,v_1) \vee +edge(e,v_1 v)) \wedge +conductivity(op,e,v,v_1)) \longrightarrow mark(e) \wedge mark(v_1)$
$mark(source(t*,v),op)$	$\forall e \in E, v_1 \in N : (type(e) = t \wedge +edge(e,v,v_1) \wedge +conductivity(op,e,v,v_1)) \longrightarrow mark(e) \wedge mark(v_1)$
$mark(target(t*,v),op)$	$\forall e \in E, v_1 \in N : (type(e) = t \wedge +edge(e,v_1,v) \wedge +conductivity(op,e,v,v_1)) \longrightarrow mark(e) \wedge mark(v_1)$

implementation *cimp* associated to the component interface *c*. To allow the operation, the implementation of the component had to exist in the architecture and therefore a corresponding node must be present in the ASCM graph. The invariant stipulates that the existence of the interface component *c* must be verified before and after the operation realization. No instance of *cimp* should be defined in the architecture $(-subComp(*,cimp))$. *cimp* should not contain any subcomponents $(-source(hasSubComp*,cimp))$. The result of the operation is the deletion of the component implementation and all of its input and output edges. In the case of at least one invariant is not respected, an impact is propagated locally to the linked nodes (components) in the ASCM graph.

Considering the distributed environment, and during software analysis, the graphs are constructed on each site and are interconnected. Then, the implementation deletion may have an impact on other nodes belonging to an ASCM graph which is distributed on a remote site. This is done by the distributed knowledge-based system.

6 CHANGE IMPACT PROPAGATION IN DISTRIBUTED SOFTWARE ARCHITECTURE

The change impact 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. This process would involve advising the user the software components to be changed and the types of the changes.

Our approach is based on the ECA formalism (Event - Condition - Action) to describe the change impact propagation rules (Hassan et al., 2009). It consists of analyzing the impact on the distributed architecture by defining generic rules. These could be applied, independently of ADLs, to propagate the impact locally from the architecture to its corresponding source codes and then to the distributed ones.

6.1 Knowledge-Based System for Change Impact Propagation

The change impact analysis is based on a knowledge-based system to manage evolution rules for distributed architecture and source codes. These rules estimate the impact of a change performed on any component belonging to the architecture or to the source codes. The knowledge-based system consists of three main components : the facts base, the rules base and the inference engine. The facts base constitutes the working memory and the dynamic part of the knowledge-based system (KBS). It contains the facts set that allows firing change propagation rules. A Fact represents a graph node or edge representing the elements of ASCM or SCSM models. These are added to the facts base during the analysis phase of the architecture specification and related source codes. Applying a change on a node is reflected in the fact base and may cause the inference engine to fire rules to perform change impact analysis.

6.2 Change Propagation Rules Definition

The change propagation rules describe formally the impact of the change operations performed on the elements of the ASCM model. The fact corresponding to the changed node or edge is updated in the knowledge-based system. This starts the change propagation process by selecting the set of rules having the updated fact in their pre-condition.

The table 2 shows the formalization of the marking assertions, which consists to identify the components and the relationships affected by a change operation. The impact propagation to the neighborhood of a marked component is defined following the nature of incoming and outgoing relationships. This refers to the impact conductivity of each relationship. The assertion $+conductivity(op,e,A,B)$ formalizes the impact conductivity of a relationship $e : A \longrightarrow B$, with A is the source component of e, B the destination component of e and op is the change operation applied on A.The marking assertions are defined as follows :

- The assertion $mark(el)$ consists to change the state of components and relationships as $affected$, in the facts base.
- The assertion $mark(v,op)$ indicates for an operation op applied on v, that the edges of v are marked as affected, according to the conductivity of the relationship.
- The assertions $mark(source(t*,v,op))$ and $mark(target(t*,v,op))$ consists to mark the relationships e of type t having v as a source node or target node and to mark the related target node or source node of each relationship e following its conductivity for the operation op.

The process of marking nodes and propagating the impact stops when all rules are fired and there is no more candidate facts.

We define three generic rules to deal with the change impact propagation :

```
$Rule\_Execute\_Operation(Op,el)\{$
    $TRUE(Op.Pre-Condition) \wedge TRUE(Op.Invariant)$
    $\longrightarrow Op.Post-Condition$
$\}$
```

The rule $Rule\_Execute\_Operation(Op,el)$ consists to perform the change operation op on an architectural element el when the pre-conditions and invariants of op are satisfied. There is no impact on the distributed architecture.

```
$Rule\_Impact\_Operation(Op,el)\{$
    $TRUE(Op.Pre-Condition) \wedge FALSE(Op.Invariant)$
    $\longrightarrow mark(el)$
$\}$
```

The rule $Rule\_Impact\_Operation(Op,el)$ changes the state of el to $affected$ in order to identify the impact of op when at least one invariant is not respected for op.

```
$Rule\_Propagate\_Impact(Op,el)\{$
    $+state(el,'affected')$
    $\longrightarrow mark(el,op)\}$
$\}$
```

The rule $Rule\_Propagate\_Impact(Op,el)$ propagates the impact to the neighborhood of an affected element el. The propagation is based on the marking assertions in order to change the state of the edges and of the linked nodes.

7 CHANGE IMPACT PROPAGATION IN DISTRIBUTED SOFTWARE ARCHITECTURE

The figure 2 presents the global architecture of our platform based on Eclipse which is extended for the change impact analysis. The extensions of our platform are divided into four main components :

The *multi-languages analyzer* allows analyzing the source codes and the architectural description of a software. The analysis is based on the grammar of each language used to write a project file. The multi-languages analyzer is based on parsers which are generated through the Java Compiler Compiler (JavaCC).

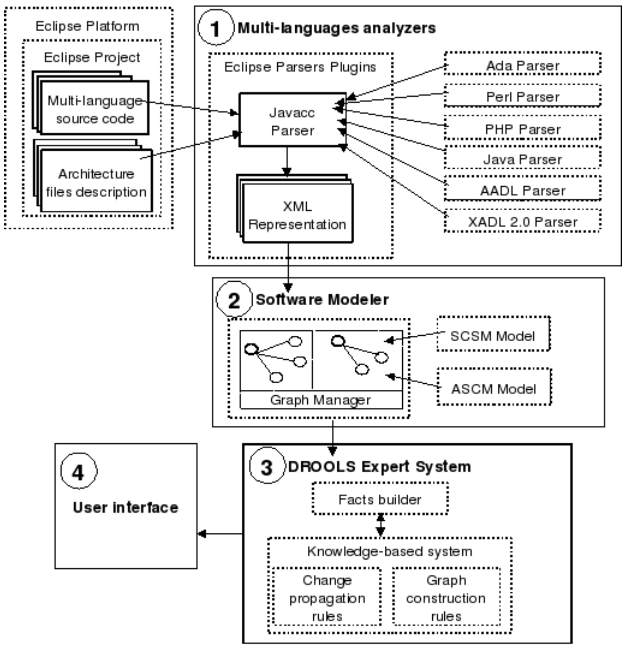


Figure 2: The architecture of the eclipse-based platform

The multi-languages analysis result consists of producing XML descriptions that are used for the graph construction by the second plugin.

The *software modeler* is the second Eclipse plugin that contains the implementation of both SCSM and ASCM models. It provides an XML flow analyzer that matches the components or relationships defined in our model with the XML tags provided by the *multi-language analyzer*. The two models are instantiated as graph where nodes represent the components and edges their relationships. The software modeler performs fact assertions in the knowledge-based system, which is the central component of our implementation and provides the change impact propagation mechanism.

The *expert system engine* is the third plugin and represents the implementation of the knowledge-based system. It allows implementing the change impact propagation process and manages the change propagation rules.

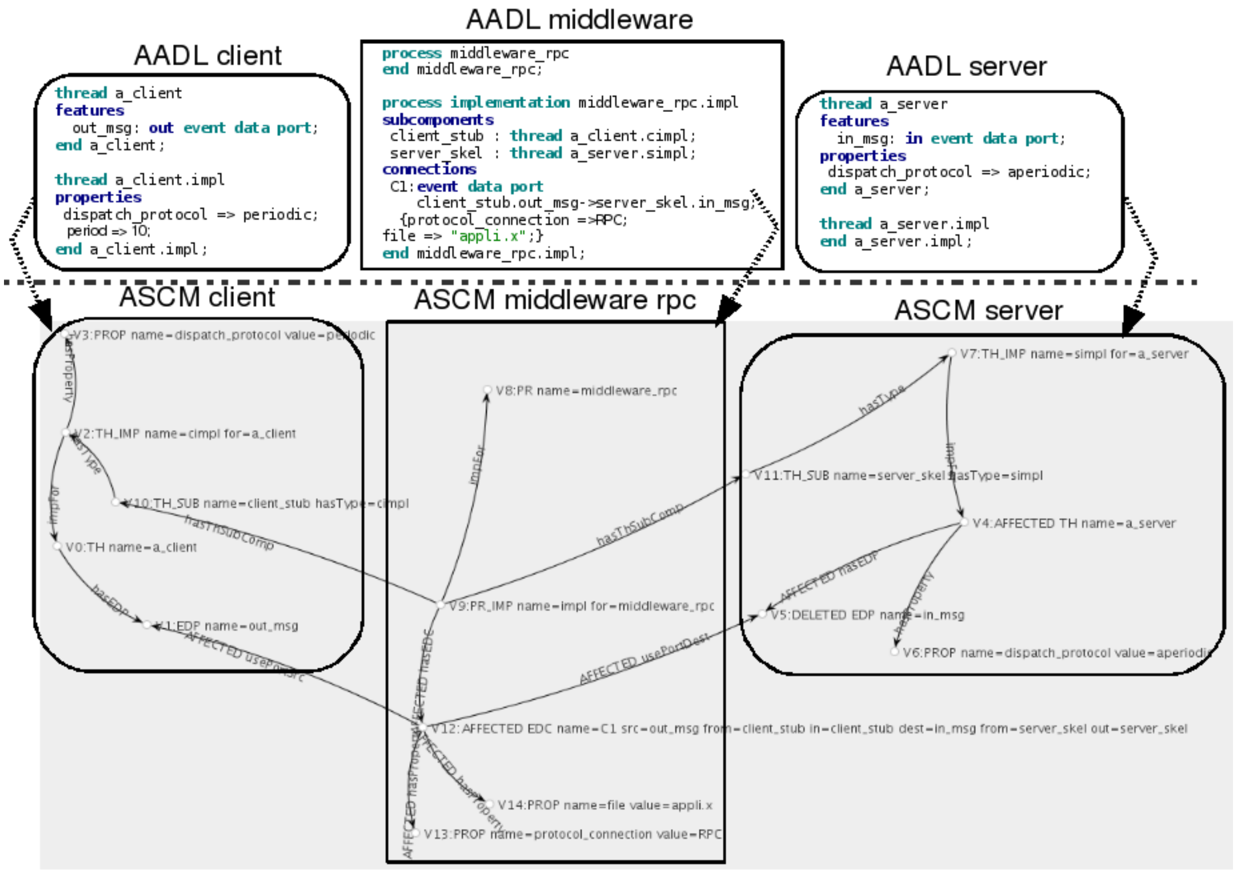
The *user interface* is the forth Eclipse plugin that allows the graph visualization and the execution of change operations on it. It allows maintainer to manage interactively the evolution and the maintenance of a software system.

Hereafter, we propose a scenario to illustrate the graph representation of a software architecture and the change propagation process.

8 CHANGE PROPAGATION SCENARIO

The proposed scenario illustrates the change impact analysis on a distributed and multi-threaded application. The application defines client threads that send messages using Remote Procedure Call protocol (RPC). The messages are consumed by threads deployed on a server. This application architecture is specified using the AADL language, presented in listings on the Figure 3.

The scenario consists to delete the server port *in_msg* using the change operations called $del\_port(a,p)$. Regarding the operation invariants and the ASCM graph, an edge exists between the node $EDPname = in\_msg$ and the node $EDCname = C1$. This edge provides conductivity of the impact of the port component. The rule *ImpactDeletePort*, described below marks and changes the state of the port node to $affected$. This fires the second rule, called *PropagateImpactDeletePort* to mark the edges entering in the port and the related target nodes. These marked nodes are sent to the Eclipse client to be inserted in the distributed fact base and which will fire the rules for propagating locally the impact. The propagation is done through relationships according to theirs conductivity. The result of the change propagation process can be shown in figure 3. The nodes and the edges, which have been marked by our expert


AADL client
AADL middleware
AADL server
ASCM client
ASCM middleware rpc
ASCM server
V3:PROP name=dispatch_protocol value=periodic
hasProperty
V2:TH_IMP name=cimpl for=a_client
hasType
V10:TH_SUB name=client_stub hasType=cimpl
implFor
V0:TH name=a_client
hasEDP
V1:EDP name=out_msg
AFFECTED usePortSrc
hasThSubComp
V8:PR name=middleware_rpc
implFor
V9:PR_IMP name=impl for=middleware_rpc
AFFECTED hasEDC
V12:AFFECTED EDC name=C1 src=out_msg from=client_stub in=client_stub dest=in_msg from=server_skel out=server_skel
AFFECTED hasProperty
V14:PROP name=file value=appli.x
V13:PROP name=protocol_connection value=RPC
hasThSubComp
V11:TH_SUB name=server_skel hasType=simpl
hasType
V7:TH_IMP name=simpl for=a_server
implFor
V4:AFFECTED TH name=a_server
AFFECTED hasEDP
V5:DELETED EDP name=in_msg
hasProperty
V6:PROP name=dispatch_protocol value=aperiodic
AFFECTED usePortDest

```
thread a_client
features
  out_msg: out event data port;
end a_client;

thread a_client.impl
properties
 dispatch_protocol => periodic;
 period => 10;
end a_client.impl;
```

```
process middleware_rpc
end middleware_rpc;

process implementation middleware_rpc.impl
subcomponents
 client_stub : thread a_client.cimpl;
 server_skel : thread a_server.simpl;
connections
 C1:event data port
    client_stub.out_msg->server_skel.in_msg;
  {protocol_connection =>RPC;
file => "appli.x";}
end middleware_rpc.impl;
```

```
thread a_server
features
  in_msg: in event data port;
properties
 dispatch_protocol => aperiodic;
end a_server;

thread a_server.impl
end a_server.impl;
```

Figure 3: Distributed Architecture evolution and related graphs resulting from the port deletion operation

system, are shown with the *affected* label. The result shows the projection relationship which leads the impact to the source codes.

```
rule "ImpactDeletePort"
when
 n : ArchitectNode(
stateNode==ArchitectNode.STATE_DELETED,
typeNode=="EDP"
     )
 n1: ArchitectNode()
 e : ArchitectEdge()
 eval((e.getNodeDest()==n && e.getNodeSrc()==n1)
      ||
      (e.getNodeSrc()==n && e.getNodeDest()==n1)
     )
then
 n.setLabel("AFFECTED "+n.getLabel());
 n.setStateNode(ArchitectNode.STATE_IMPACTED);
end

rule "PropagateImpactDeletePort"
when
 n : ArchitectNode (
stateNode==ArchitectNode.STATE_IMPACTED,
typeNode=="EDP"
     )
 n1: ArchitectNode()
 e : ArchitectEdge()
 eval((e.getNodeDest()==n && e.getNodeSrc()==n1)
      ||
      (e.getNodeSrc()==n && e.getNodeDest()==n1)
     )
then
 e.setState(ArchitectEdge.STATE_IMPACTED);
 n1.setStateNode(ArchitectNode.STATE_IMPACTED);
end
```

9 CONCLUSIONS AND FUTURE WORKS

In this paper, we have presented a ASCM model and a distributed knowledge-based system approach to deal with the change impact analysis on distributed software architecture. Our model represents the informations extracted from architectural descriptions, independently of the ADLs, as distributed graphs which are stored as facts in the distributed knowledge-based system. The facts fire the change propagation rules in order to identify the impact of a change and to propagate it to linked nodes. Our approach is implemented as many plugins in Eclipse Environment.

Perspectives of our work consists to enhance our change impact analysis approach to deal with qualitative aspects of a software at the architecture level.

REFERENCES

Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison Wesley.

Clements, P., Kazman, R., and Klein, M. (2002). *Evaluating Software Architectures: Methods and Case Studies*. Addison Wesley.

Clements, P. and Shaw, M. (2009). "the golden age of software architecture" revisited. *IEEE Software*, 26:70–72.

Deruelle, L., Basson, H., Bouneffa, M., and Hattat, J. (2007). An eclipse platform extension for analysis and manipulation of multi-language software code. pages 174–179.

Deruelle, L., Bouneffa, M., Melab, N., and Basson, H. (2001). A change propagation model and platform for multi-database applications. *Software Maintenance, IEEE International Conference on*, pages 42–51.

Garlan, D., Monroe, R., and Wile, D. (1997). Acme: An architecture description interchange language. In *in Proceedings of CASCON97*, pages 169–183.

Hassan, M. O., Deruelle, L., and Basson, H. (2009). Towards a change propagation process in software architecture. In *18th International Conference on Software Engineering and Data Engineering (SEDE-2009)*, pages 85–90, Las Vegas, Nevada, USA.

Medvidovic, N. and Taylor, R. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26:70–93.

Mens, T. (2001). Transformational software evolution by assertions. *Workshop on Formal Foundations of Software Evolution, CSRM 2001*.

Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing.