DYNAMIC RECONFIGURABLE SOFTWARE ARCHITECTURE:
ANALYSIS AND EVALUATION

Amar RAMDANE-CHERIF [1] , Nicole LEVY [1] and Francisca LOSAVIO [2]

1 *PRiSM, Université de Versailles St.-Quentin,*
45, Avenue des Etats-Unis, 78035 Versailles Cedex, France.
{ amar.ramdane-cherif@prism.uvsq.fr}

2 *Centro ISYS, LaTecS, Universidad Central de Venezuela,*
Caracas, Venezuela.

Abstract: **Dynamic changes to an architecture is an active area of research within the software architecture community. Architectures must have the ability to react to events and perform architectural changes autonomously. In this paper, we focus on dynamic architectures reconfiguration. Our principle is to use the agent architectural concept to achieve this functionality with respect to some quality attributes. Hence the questions that we are currently facing: what are the architectural principles involved in building adaptable architecture? How should these architectures be evaluated? In addition, we adopt the B formal method to support design specifications for agent software architecture. Formal modeling of a specification of our agent software architecture enables us to analyze and reason about it with mathematical precision and allows obtaining the abstract specification of the initial architecture formally. Besides, the design decisions are stored with the goal of making the reconfiguration tasks easier by the agent. This paper describes work in progress and presents some interesting ideas connected to architectural agents.**

Key words: **Software architecture, Quality attributes, Formal method.**

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35607-5_15

J. Bosch et al. (eds.), *Software Architecture*

1. INTRODUCTION

A critical aspect of any complex software system is its architecture. The "architecture" term conveys several meanings, sometimes contradictory. In our research we consider that architecture deals with the structure of the components of a system, their interrelationships and guidelines governing their design and evolution over time [1][2]. The architectural model of a system provides a high level description that enables compositional design and analysis of components-based systems. The architecture then becomes the basis of systematic development and evolution of software systems. Furthermore, the development of complex software systems is demanding well-established approaches that guarantee the robustness and other qualities of products. This need is becoming more and more relevant as the requirements of customers and the potential of computer telecommunication networks grow. A software architecture-driven development process based on architectural styles consists of a requirement analysis phase, a software architecture phase, a design phase and maintenance and modifications phase. During the software architecture phase which we present in figure -1-, one models the system architecture. To do so, a modeling technique must be chosen, then a software architectural style must be selected and instantiated for the concrete problem to be solved. The architecture obtained is then refined either by adding some details or by decomposing components or connectors (recursively going through modeling, choice of a style, instantiation and refinement). This process should result in an architecture that is defined, abstract and reusable. The refinement produces a concrete architecture meeting the environments, the functional and non-functional requirements and all the constraints on dynamics aspect besides the static ones.

Fortunately, it is possible to make quality predictions about a system. These will be based solely on an evaluation of its architecture. However, it is important to provide a method operating at the architectural level that will provide a substantial help in detecting and preventing errors early in development. We are interested in applying the previous software architecture phase to provide a new approach based on an architectural agent. Such an agent is used to supervise the architecture, gather information from it and its environment, capture dynamic changes, and manage them. it monitors the components dynamically and adapts them to structural changes in the architecture. The correctness and robustness of the architecture is ensured by the agents as the changes take place so that the system conforms to its architecture and remains in conformance throughout its lifetime. The B formal method will be used to specify precisely the structure and the

behavior of our architecture and to prove rigorously that this architecture satisfies the desired structural and behavioral properties.

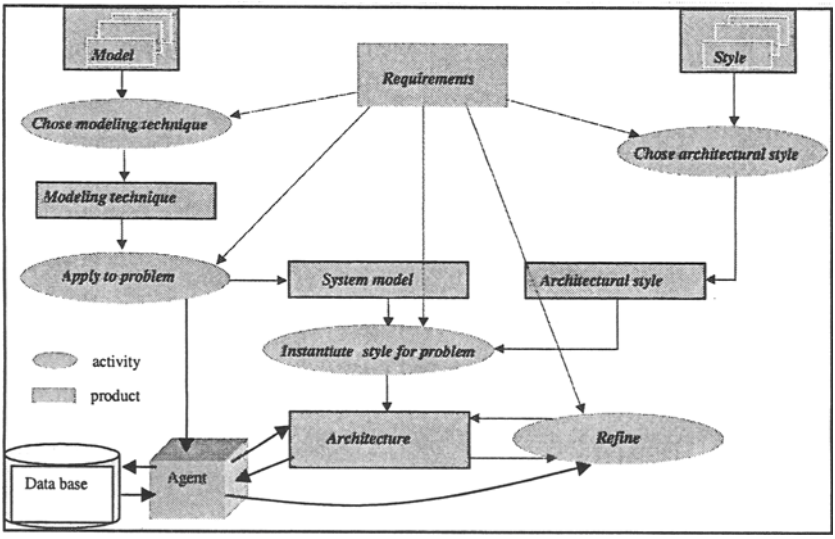

Figure 1. Software architecture phase

This paper is organized as follows. In the next section, we will introduce the related work and then our approach and some ideas about its methodology and framework will be presented. Then, we will briefly describe the B formal method used to specify our architecture. In the next section, we describe an application which is highly simplified for presentation purpose. Finally, the paper concludes with a discussion of future directions for this work.

2. RELATED WORK

In earlier works on description and analysis of architectural structures the focus has been on static architectures. Recently, the need of the specification of the dynamic aspects besides the static ones has increased [3][4]. Several authors have developed some approaches on dynamism in architectures which fulfill the important separation of the dynamic reconfiguration behavior from the non-reconfiguration. These approaches increase the reusability of some systems components and ease the understanding. In [5], the authors use an extended specification to introduce dynamism in Wright-language. The work in [6] focuses on the addition of a complementary

language for expressing modifications and constraints in the message-based C2-architectural-style. A similar approach is used in Darwin [7] where a reconfiguration manager controls the required reconfiguration using a scripting language. Many other investigations have addressed the issue of dynamic reconfiguration with respect to the application requirements. For instance, Polylith [8] is a distributed programming environment based on a software bus which allows structural changes on heterogeneous distributed application systems. In Polylith, the reconfiguration can only occur at special moments called reconfigurations points explicitly identified in the application source code. Thus, this mechanism presents some disadvantages making Polylith unsuitable for the purpose of dynamic reconfiguration. The Durra programming environment [9] supports an event-triggered reconfiguration mechanism. Its disadvantage is that the reconfiguration treatment is introduced in the source code of the application and the programmer has to consider all possible execution events which may trigger a reconfiguration. Argus [10] is another approach based on transactional operating system then the application must comply to a specific programming model. This approach is not suitable to deal with heterogeneity and interoperability. Conic [11] approach proposes an application independent mechanism where reconfiguration changes affect component interactions. Each reconfiguration action can be fired if and only if components are in a determined state. The implementation tends to lock a large part of the application, hence, causing important disruption. New formal languages are proposed for the specification of mobility features; a short list includes [12] and [13]. Particularly in [14] a new experimental infrastructure is used to study two major issues in mobile component systems. The first issue is how to develop and to provide a robust mobile component architecture and the second issue is how to write code in these kinds of systems. This analysis makes it clear that a new architecture that permits the dynamism reconfiguration, adaptation and evolution while ensuring the integrity of the application is needed. In the next section, we propose such an architecture based on agent components.

3. AGENT SOFTWARE ARCHITECTURE

Our idea is to include additional special intelligent components in the architecture called "Agents". The agents act autonomously to adapt dynamically the application without requiring outside intervention. Thus, the agents monitor the architecture, perform reconfiguration, evolution and adaptation, to structural changes at the architectural level and achieve effective reactive architectural concept as shown in figure-2(a)-

3.1 Agent interface

The interface of each agent is defined as the set of provided actions but also required events. To each agent we attach Event/Condition/Action-rules mechanism in order to react with the architecture and the architectural environment and perform activities. Performing an activity means invoking one or more dynamic method modification with suitable parameters. Figure-2(b)- provides a schematic overview of an agent.

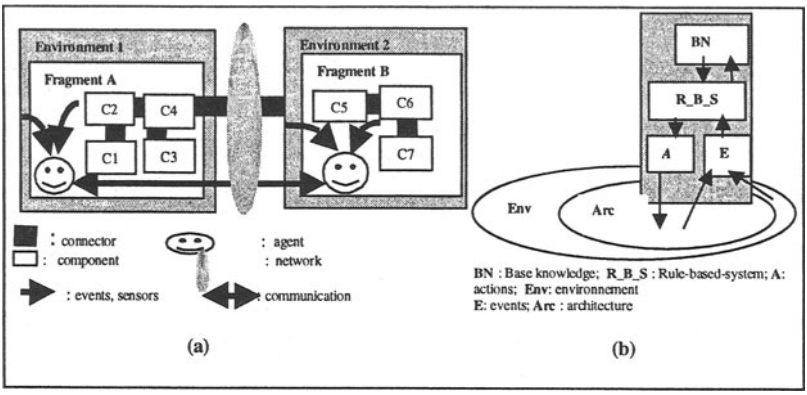

Figure 2. (a) The based architecture; (b) Schema overview of an agent

3.2 Agent Knowledge

The agent has a complete knowledge of the architecture or simply of the configuration part of the architecture that implements one relevant aspect. However, the agent can obtain information about other parts of the architecture by communicating with others agents. The agent provides the architectural operations needed to build up, add, delete, modify (faulty data), update, adapt, assembly, check (for new version), immigrate, transfer, restart and ...(etc) a specific component, connector or a configuration. The agent implements several different protocols of dynamic switching of architectures. All the structuring architectural information and the full definitions of all the protocols are its part of knowledge base.

Therefore, the agent is the locus of dynamic topological transformations, it constructs an initial topology at system's start-up and provides a set of topological operations to modify it.

3.3 Agent rule

The behavior of an agent is expressed in terms of Rules which are grouped in the behavior units. The concept of behavior units is used to partition the behavior of an agent. Each behavior unit belongs to one class of the architecture modification and is associated with specific triggering event type. At reception of some event of this type, the behavior described in this behavior unit is activated. The event is defined by name and the number of parameters. For instance, **check** (object) is a notification event whose name is check and has one parameter object. In this protocol, the agent receives the events which are expressions over names and the parameters of a notification. So, for example **check*** (_,_) would match all the notifications whose name starts with check and that have two parameters. The body of a behavior unit is a set of dynamic rules having the form:

IF "Preconditions" THEN "Actions"

The preconditions of a rule are expressed as a Boolean formula that have to be satisfied before the actions described in the THEN part can be executed. The receipt of a triggering event by a behavior unit activates all the dynamic rules of that behavior unit. The preconditions of rules of the same behavior unit are mutually exclusive, so that exactly one of the rules will always be fired. Actions in the THEN part of a rule may modify/create/delete ....components/connectors instances and/or produce some events sent to other behavior units or to the external architecture and its environment. The dynamic behavior of each object class modification is modeled as a collection of rules grouped in behavior units specified for that class and triggered by specific events.

In the following we give a brief description of the B formal method that we used to specify our architecture dynamic services needed for reconfiguration, adaptation and evolution actions.

4. THE B FORMAL METHOD

B is a formal method developed by Abrial [15]. It is a complete method that supports a large segment of the development life cycle: specification, refinement and implementation. It has already been used in significant industrial projects and commercial case tools are available in order to help the specifier during all development process. In the B method, the are three syntactic kinds of components: abstract machine, refinements and implementation. In our work we have used the B method for specifying, designing and coding our Agent component as shown in Figure-3-.

- First, a high level of abstraction is used for the initial specification which abstracts from the details and describes the observable behavior of our agent architecture and the global view of the functionality that it provides. Then, explicit proof obligations are provided. Proof of these obligations ensures that the relevant properties of the system hold.
- Second, a refinement allows us to gradually add more detail to our previous abstract specification. Explicit proof obligations for refinement are provided. Proving these obligations ensures that the relevant properties of the system still hold.
- Third, an implementation is the last level of a development, it cannot be refined, so it can be translated into code.

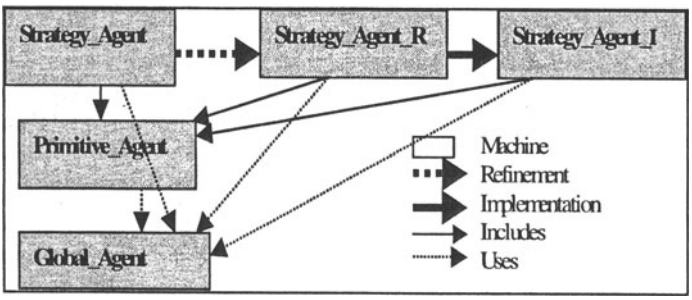

Figure 3. The agent abstract machine, refinement and implementation

In the following, we give through a simple application, just a part of our definition of the specification due to space limitations.

5. APPLICATION

In this section, we describe our application. It is a simple distributed shared to-do list application in which client and manager share a list of queries. This application, which is highly simplified for presentation purposes (figure-4-), consists of :

COMPONENTS :

1. The visualizer component displays for user the current contents of a shared list. It has three ports: the first port (*V_provide_port)* connects to a shared list component, the second port (*V_required_port1*) receives events which indicate changes in a shared list, and the third port (*V_required_port2*) shares the currently marked entry in the list with any interested component.

2. The editor component has two ports (*E_required_port1*; *E_required_port2*) which connect it to a shared list and to marked entry of the visualizer component. The user can add new entries to the list or edit other selected entries in the list.
3. The delete button component is connected to a shared list and to a marked entry and if pressed, it deletes the marked entry in the list. So it has two ports (*Delete_required_Port1*; *Delete_required_Port2*)
4. The done button component is connected like the delete button. When pressed, it sets the flag of the marked entry to "completed". It also has two ports (*Done_required_Port1*; *Done_required_Port2)*
5. The shared list component resides on a server and maintains a list of queries. This list is shared via the *ToDoList_provide_port* port and other components are notified of changes via the *ListChanged_provide_port* event port.

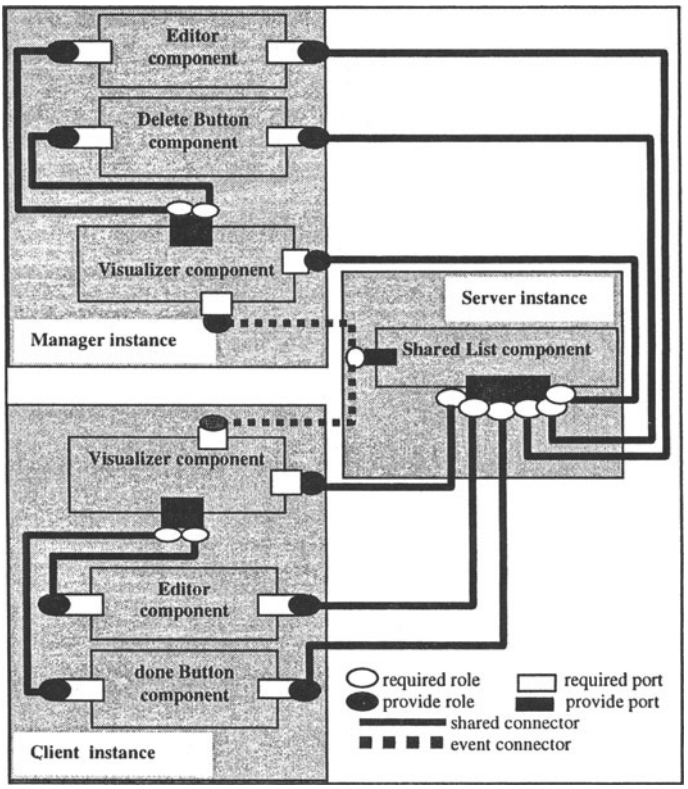

Figure 4. A simple distributed shared to-do list application

CONNECTORS:

1. A shared connector has two roles: the *S_provide_role* and the *S_required_role*. Such a connector is in charge of connections between the components (editor, visualizer, delete button and done button) and the shared list component and between the visualizer and the editor. For example one shared connector associates his role *"S_required_role"* with the port *"ToDoList_provide_port"* of the shared list component and his role *"S_provide_role"* with the port *"V_required_port1"* of the visualizer component.
2. An event connector has two roles: the *E_provide_role* and the *E_required_role*. Such a connector is in charge of connections between the visualizer and the shared list component. It associates his role *"E_provide_role"* with the event port *"ListChanged_provide_port"* of the shared list component and his second role *"S_provide_role"* with the port *"V_required_port2"* of the visualizer component.

These components and connectors are used to compose a distributed shared to-do list application. This application is distributed over three locations. The shared list component instance resides on a server and is connected to an instance of the manager on one machine and to an instance of the client on another machine. The client instance contains a visualizer component, an editor component and a done button component. The client may only see the contents of the list and mark entries as "done " by pressing the "done" button. The manager instance contains a visualizer component, an editor component and a delete button component. The manager can actually add new entries to the list and delete them.

According to the requirements of the application, security quality attribute is more important than other quality attributes. Hence, we assume now that this application has to be extended with a security component (figure-5-). This component will encrypt the data exchange between the client and the server.

Security : it is a measure of the system's ability to resist to unauthorized attempts of usage and denial of service while still providing its services to legitimate users. At the architectural level:

- It means to have a mechanism or device (software or hardware). It may be a component or integrated into a component.
- It is measured by an attribute with Boolean value, depending on the presence or not of a mechanism or a device.

In order to provide the security quality attribute of the architecture mentioned above, a modification to this architecture must be performed stepwise by the agent.

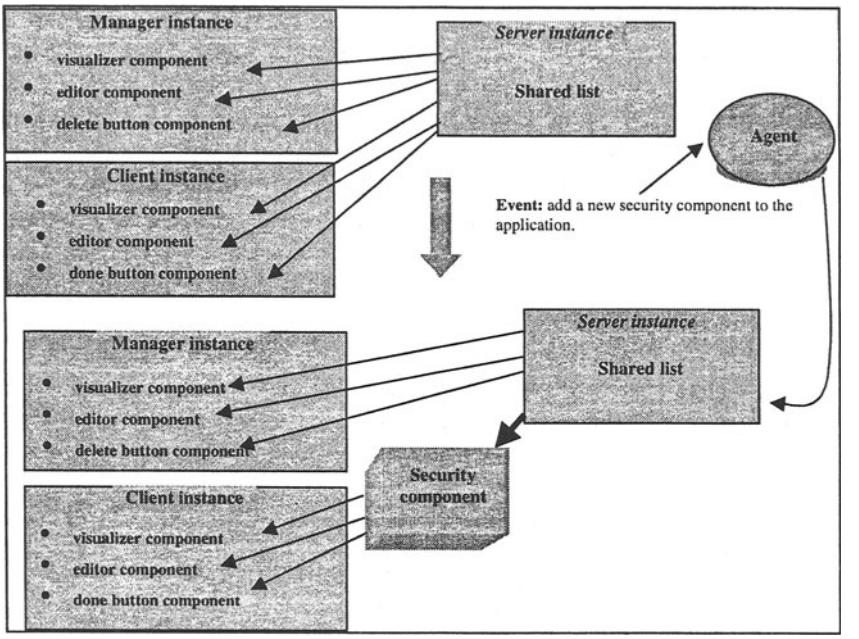

Figure 5. The client component is extended with a security component

In the following we give just a part of our definition of the specification due to space limitations. The specification concerns only the agent machine. The analysis of our agent_architecture consists in studying its statics and dynamics. The statics corresponds to the definition of the state whereas the dynamics corresponds to that of the operations.

5.1 The agent static part

The static part of the agent based architecture (**Primitive_Agent** machine) contains the formalization of the architectural representation which is based on generic components, connectors, configurations, ports, roles and bindings.

The definitions of types are formalized in a B machine **Global_Agent.** In this machine the clause **SETS** give the sets used to formalize the agent architecture. These sets are considered as basic independent types. Such sets can be enumerated or deferred (a finite and non-empty unspecified set). The **VARIABLES** clause of the **Primitive_Agent** machine introduces the variables of the state of the agent architecture and the **INVARIANTS** clause its invariant. The invariant is defined in terms of the variables by means of

the formal languages of predicate calculus and set theory. It consists of a number of predicates separated by the conjunction. The variable of the machine consists of some sets. The invariant of the machine contains both the typing of each of the variable and several relations or functions representing the relationship between them. The invariant clause contains also several predicates expressing architectural constraints and assumptions containing in the knowledge base of the agent.

5.2 The agent dynamic part

The dynamics of the agent based architecture (machine **Strategy_Agent Primitive_Agent**) is expressed through its operations. The role of an operation, as later executed by the computer, is to modify the state of the abstract machine, and this, of course, within the limits of the invariant. The clause **OPERATIONS** of the **Primitive_Agent** machine is made up of the primitive operations and that of the **Strategy_Agent** machine of the composite operations which call upon the operations of the **Primitive_Agent** machine. The **Strategy_Agent** machine includes the **Primitive_Agent** machine. Each operation (Rule) has the following syntax.

Name-operation(parameters) =
PRE
pre-conditions
THEN
Actions (instructions)
END ;

The operations of the machine consists of:

a) For Primitive_Agent machine: create component, connector, role and port, add port to components, add role to connectors, create connection, get a value of quality attribute of a component, get a value of quality attribute of a connector, set quality attribute value for a component, set quality attribute value for a connector .....
b) For Strategy_Agent machine: add component to an architecture, add connector to an architecture, delete component from an architecture, delete connector from an architecture, delete connection from an architecture, get quality attribute value of the global architecture, transfer state component, migrate component, ...

All theses operations are used by the agent for changing the architecture dynamically. The machine **Strategy_Agent_r** will refine the **Strategy_Agent** by adding some details about some operations. Finally the final machine is an implementation machine **Strategy_Agent_i**. This machine will transform the abstract model of our architecture into another

model that is all concrete. Using the Atelier B we will provide explicit proof obligations of the abstract machine and we will prove these obligations to ensure that the relevant properties of the system hold. Explicit proof obligations for refinements machines will also be provided and proved to ensure that the relevant properties of the architecture hold in the refinements. The last refinement which is an implementation machine will be translated into code source (figure-6-).

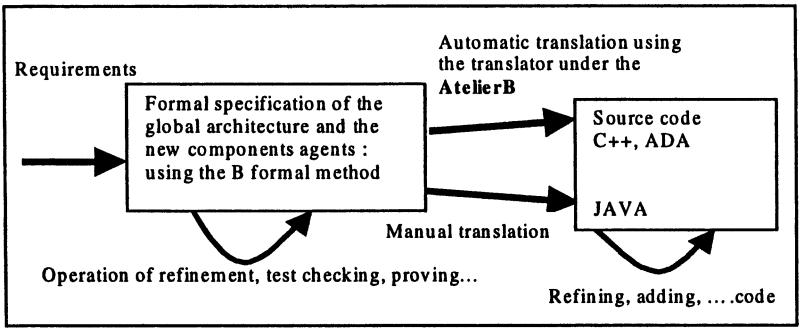

Figure 6. Generation of source code

5.3 Configuration mechanism and evaluation of quality attributes

In order to be able to evaluate the quality attributes of an architecture, a set of variables representing them have been introduced within the **Global_Agent** B machine. These variables are defined by functional expressions. In the **INVARIANTS** clause of the **Primitive_Agent** machine, the attributes are constrained by predicate expressions. Therefore, it becomes possible to measure the impact in terms of a quality attribute on an architecture by applying some operation presented in the clause **OPERATIONS**. It remains to describe the modifications strategies allowing the enhancement of one specific quality attribute. These strategies are not formalized for the moment. But they could be included in the agent knowledge base. The reconfiguration must done in safe way to ensure at the execution time the integrity of the global architecture.

Event: a new security component has to be added to the application (between the client and the server).

The event received by the agent can be:

a) The user's event that manages the system and asks for an evolution of the architecture toward a more elevated security level.

b) An event of the system and the environment that the agent controls. The agent can test the measure of the security attribute. The agent is able to test the presence of the security component in the application and to take the correct decisions.

The agent will use the following strategy which consist to apply some rule operations (figure 7):

1. Create a new security component, and add ports to it (2 provide and 2 required).
2. Create connectors (4 shared connectors and 2 event connectors), add a special required and provided roles to each connector, and create connections between the ports of the security component and respectively the client component and the server component via the appropriate connectors.
3. For each old connection between the server component and the client component, test if the corresponding connector is passive then delete this connection and transfer the state of the corresponding connector to the new connector already created via security component.

6. CONCLUSION

The main contributions of this paper can be resumed as follows. We have suggested to use the B formal method to model the possible adoption of adaptive based agent paradigms in software architecture. Formal modeling of a specification of a software architecture provides an unambiguous representation. This representation allows for rigorous analysis and reasoning of both functional properties and quality attributes. However, we are providing a methodology that, starting from a set of B specifications, derives a performance model that allows the designer early in the design phase, to evaluate the software architecture. The agents have the ability to react to events and perform architectural changes autonomously. We are currently experimenting on application examples of how agents can be introduced and how they improve the security quality attribute of a distributed system. We have given ideas about the reconfiguration, adaptation and evolution of the proposed architecture. However, there are some issues that we have not dealt with in this paper. We have developed our abstract specification using the B method. This specification contains the formalization of the architectural representation, the architectural constraints, the agent knowledge semantics and all operations used by the agent for changing the architecture dynamically. The passage from this specification to implementation throw refinements are undertaken. These refinements will be carried out entirely under the control of the Atelier B tool and will be

concluded by some proofs to ensure that the relevant properties of our architecture hold.

References

[1] M. Shaw, D. Garlan, Software Architecture, Perspectives on Emerging Discipline, Prentice-Hall, Inc. , Upper Saddle River, New Jersey, 1996.

[2] D. E. Perry, A. L. Wolf, Foundations for the Study of Software Architecture, Software Engineering Notes, 17(4):40, Oct. 1992.

[3] P. Oreizy, N. Medvidovic, R. N. Taylor, Architecture-Based Runtime Software Evolution, Proc. 20 th Int'l Conf. On Soft. Eng. (ICSE'98), pp. 177-186, Kyoto, Japan, Apr. 1998.

[4] P. Ciancarini, C. Mascolo, Software Architecture and Mobility, In the Third International Software Architecture Workshop(ISAW-3). Page 1-4, ACM-Press, Orlando Florida, Nov. 1998.

[5] R. J. Allen, R. Douence, D. Garlan, Specifying Dynamism in Software Architectures, In Proceedings of Foundations of Component-Based Systems Workshop, Sep. 1997.

[6] R. N. Taylor, N. Medvidovic, K. N. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, D. L. Dubrow, A Component and Message-Based Architectural Style for GUI Software, IEEE Transactions on Software Engineering, Jun. 1996.

[7] J. Kramer, J. Magee, Self Organizing Software Architectures, In Joint Proceedings of the second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints'96). Pages 35-38, ACM-Press, 1996.

[8] J. M. Purtilo, The Polylith Software Bus. ACM TOPLAS, 16(1):151-174, 1994.

[9] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, R. Lichota, Durra: A Structure Description Language for Developing Distributed Applications, IEEE Software Engineering Journal, pages 38-94, mar. 1993.

[10] T. Bloom, M. Day, Reconfiguration and Module Replacement in Argus: Theory and Practice, IEEE Software Engineering Journal, pages 102-108, mar. 1993.

[11] J. Kramer, J. Magee, Dynamic Structure in Software Architectures, In Proceedings of the fourth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'96), pages 3-14, ACM-Press, Oct. 1996.

[12] R. DeNicola, G. Ferrari, R. Pugliese. KLAIM: A kernel Language for Agents Interaction and Mobility, IEEE Transactions on Software Engineering, 24(5):315-330, 1998.

[13] P. Ciancarini, C. Mascolo, Software Architecture and Mobility, In the Third International Software Architecture Workshop(ISAW-3). Page 21-24, ACM-Press, Orlando Florida, Nov. 1998

[14] W. Van Belle and J. Fabry, Experience in Mobile computing/ The Cborg Mobile Multi-Agent System, Technology of Objet-Oriented Languages and Systems, Tools 38, pages 7-18, march 2001, Zurich, Switzerland

[15] J. R. Abrial, The B Book: Assigning Programs to Meanings, Cambridge University Press, 1996

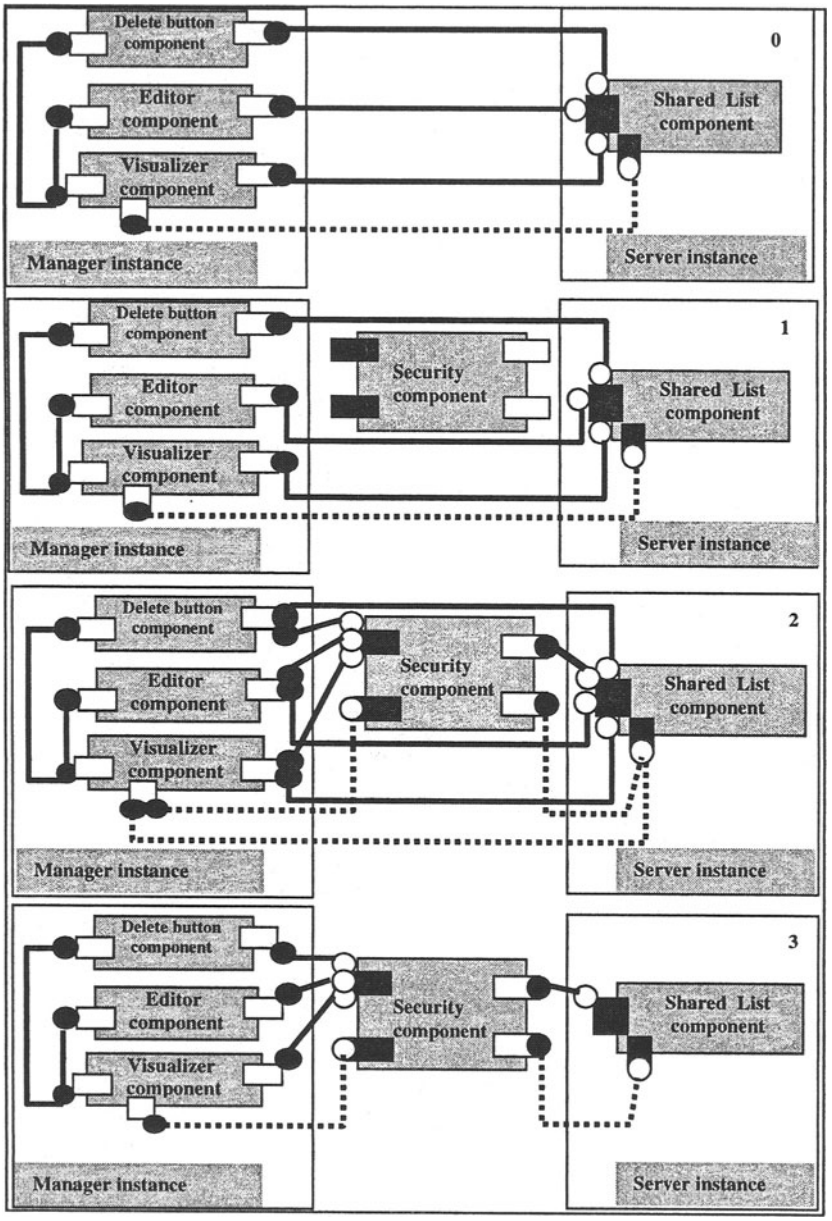

Figure 7. The different steps of architecture reconfiguration executed by the agent component