Middleware for Improving Security in a Component Based Software Architecture

Abraham Blanca[1], Aguilar Jose[1], Leiss Ernst[2]
University of Los Andes[1], University of Houston[2]
blancataz@gmail.com, aguilar@ula.ve, coscel@cs.uh.edu

Abstract

We developed a framework for reflective middleware that monitors security capabilities that every software component has. The main idea is to monitor the applications, changing or tuning the software components with the explicit goal of making the whole system as secure as possible. The middleware is flexible enough to be configured with the specific needs of the system that is going to be monitored. This experience can be implemented not only for security purposes but also for performance monitoring, load balancing or resource management in areas like Grid computing, DataWarehouses, Webservices among others.

1. Introduction

The main idea of this work is the implementation of a middleware to monitor (called introspection in [6,7]) base applications, in order to change or tune (called intercession in [6,7]) the software components with the specific goal of making the whole system as secure as possible. Levels of security are defined before start the execution of a given application, so that in runtime the middleware can make the following decisions. : tune one or more components, change one or more component, or send an alarm. When the middleware tunes a component, it changes the encryption mechanisms; if the middleware needs to change the component, then it downloads a new component and makes the change; and finally, the middleware can send an alarm to a group of users or system administrators. Tuning and changing are performed when the middleware finds a security thread in one component and its related or depended components. In this way, our middleware not only audits automatically the security of a system that is composed by several software components, but also makes changes to components that does not pass the security test. This is particularly important for big systems that have many software components, some of them developed as black boxes. Other similar works are: GridKit Middleware [9], which is one of the dynamically configurable middleware families that have been developed using the OpenCOM component model and it is oriented to manage grid resources[9]. Other work is DynamicTAO that is primarily targeted for static hard real-time applications such as Avionics systems. This one assumes that once it is initially configured, its strategies will remain in place until it completes its execution. There is very little support for on-the-fly reconfiguration. [28]. Universally Interoperable Core is a reflective middleware infrastructure that is customizable to ubiquitous computing scenarios [28]. Gaia is a component-based operating system based on a reflective middleware substrate [28]. The second part of this paper will present general theoretical aspects, the third part will explain the middleware architecture, fourth part will show the implementation, and finally, the conclusions of this work.

2. Theoretical Aspects:

2. 1. Autonomic Systems

An autonomic environment proposes the capability of self-managing systems, which must be able to have knowledge of their components, that is, status, capabilities, etc. An autonomic system is aware of its environment conditions and the context surrounding its activities. This may include the possibility of proactively changing or predicting behaviors. All this provides opportunities for planning and affecting the state of the system if this is needed [3]. Autonomic system characteristics are being applied in four fundamental areas [3]: (i) Self-configuring capabilities: adapt it to unpredictable conditions by automatically changing its configuration. (ii)Self-healing capabilities: Prevention of and recovery from failure. (iii) Self-Optimizing capabilities: Continued system tuning. (iv) Self-Protecting capabilities: Identifying and defend it

978-0-7695-3322-3/08 $25.00 © 2008 IEEE
DOI 10.1109/NCM.2008.261

against various types of attacks, such as viruses, unauthorized access, and denial of services attacks. Some of the most interesting research areas in autonomic computing are [3]: (i)Recovery operations (fault tolerance): These types of systems should find a way to stabilize themselves if a failure occurs. (ii) Predictive capabilities and continuous optimization: An autonomic system should take actions proactively to support its system objectives without user control or intervention. (iii)Security vulnerability determination: The system will protect itself from accidental and malicious harm; when damage occurs, the system should attempt to recover it. An architecture for autonomic computing includes four main aspects: the processes definition which describes the business processes that are automated in the autonomic system; the resources definition which describes the resource types that will be used and managed by the autonomic system; the technical reference architecture that describes how the system elements are going to be integrated together in order to support the services delivered in an organization; and the application patterns that are in charge of constructing templates to make all parts working together, specifying predefined situations commonly found in real deployments [5].

2.2 Software Architecture [2]

The discipline of software architecture proposes a new set of concepts for describing and reasoning about software composition at the high level of abstraction at which software architects conceive and reason about software systems. In other words, the software architecture should represent a high-level view of the system revealing the structure, but hiding most implementation details. Abstractly, software architecture involves the description of: elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. More specifically, an architectural description of a software system should identify: the partition of the overall functionality into components; the behavior of these components; the protocols used by the components to communicate and cooperate, i.e., which connectors exist between them. There are many models for software architecture, the one that was used as inspiration for this middleware is Fractal [10] which was created as a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure systems and applications, more details are provided in [10, 17]. Others very known software architectures are the three tiers architecture, five tiers architecture or n-tiers architecture, the tiers are also called layers or views and among the most common layers are the Presentation Layer, Business Tier, Business Logic, Data Access Tiers Data Tier, Logical and Physical Layers[2,22].

2.3 Computational Reflection

Reflection is the ability of a running program to examine itself and its software environment and to change what it does depending on what it determines in this process [6, 7]. Using this capability; systems can explain their behavior and modify their processing methods, for example, to improve their performance. A reflective system has two levels, a base level which consists of base entities that perform the usual functionalities of the system, regardless of whether it is reflective or not, and a meta level which consists of entities that perform reflection (Internal states and behavior can be accessed and modified through a self-representation). A reflective system processes information about itself and its environment, making changes in order to reach some goals (for example, improve its performance) [7, 10, 11]. This activity involves three important aspects: introspection (state observation), intercession (alteration of its execution and behavior), and reification (making implementation information available to the application) [7, 12, 13]. Some works with the objective of improving and creating new reflective programming languages are the following: Apertos [14, 16] is a distributed object oriented system that uses reflection.. Iguana is another project whose goal it is to provide support for the construction of (system) software that can be dynamically customized to achieve non functional changes [18, 20]. Another reflective language is OpenC++, a version of C++ with a Metaobject Protocol (MOP) [18, 20]. MPC++ is a compile-time metalevel architecture in C++ that extends and modifies language semantics, incorporating reflection properties into the C++ language [19, 20]. OpenJava is a new macro system developed for Java; its main idea is to use metaobjects, thereby incorporating into this programming language reflective computing [12,13]. Proactive is a library created for Grid resources management, it has been developed in Java and incorporates reflection among many other functionalities [1]. ABCL/R3 [21], Smalltalk and NeoClasstalk [23] are other reflective programming languages.

2.4 Multi Agent Systems (MAS)

A MAS is a system composed of several agents. An agent is a physical or abstract entity that can perceive

its environment using sensors, evaluate the perceptions in order to make decisions and communicate with others agents. In our case, an agent is defined as a software entity that has knowledge about a particular problem, can interact with other agents, and perform tasks to solve that problem [8, 15]. The agents have autonomy, so they make its own decisions and have their own resources. The agents have their local views so each agent has a limited view of the system and they work in an asynchronous and decentralized way, there is no controlling agent. Multi-agent systems can manifest self-organization and complex behaviors even when the individual strategies of all their agents are simple [8, 15].

3. The Reflective Collective Middleware

3.1 Reflective Middleware Architecture

This is an architecture composed of three intelligent agents, in order to monitor and change composed systems that are running in a known environment; the main components of the middleware are shown in Fig 1. An initial configuration is required; this is done using a XML file, where the specific tasks and the expected behaviors which are going to be monitored are defined. The three intelligent agents are defined as followed: *Monitor:* Its main goal is to observe (introspection) the base level in order to reflect on it. This agent supervises base level components, their state variables, and performances, among other issues, in order to allow system reflection. *Reflector:* This agent processes and analyses base level information. Its main function is to manage the environment variables and to interrupt direct communication among components of the system. In addition, it can generate the necessary changes (intercession) at the base level. *Information Manager:* It manages information related to the patterns, and knowledge of the system. Depending on the complexity of the information that will be used, this component uses a database or text files to store the data. All this is configured when the middleware is installed. In this way, our middleware incorporates the following characteristics into the base level: *Self-Awareness:* The system knows itself and is aware of its states and behaviors. *Self-Optimizing:* It will detect system degradation and intelligently make changes to avoid dangerous situations. *Preventive plans:* It will detect potential problems and reconfigure the system in order to keep it working. It will display proactive behavior. *Contextually Aware:* It is aware of the full system execution environment, and is able to react to changes in this environment. *Portability:* It is portable across multiple software architectures.

3.2 Reflective Middleware Library

A middleware was built to work as another layer into the system architecture of any application. This provides the possibility of monitoring the environment where the base software is loaded, which system components are active, what security risks and faults can be expected, and what decisions must be made if either an expected or unexpected issue occurs. Three main agents have been created: a Monitor, a Reflector and an Information Manager; the middleware has a *coordinator mechanism* based in direct and indirect communication, the direct communication is established using messages between agents, and the indirect communication is created when the agents go to the historic information that is manipulated by the Information Manager. That is, the communication among middleware agents is based on direct messages and blackboard indirect communication using a collective memory. Each middleware agent has well defined tasks and all their experiences are kept in XML files that construct the knowledge base. Even when three main agents exist, other specialized agents can be created to manage special cases. That is, new agents can easily be incorporated to the middleware in order to add new functionalities, when more specialized tasks want to be performed. In this case for example, a special agent was created in order to manipulate security subjects, like the encryption mechanisms. The environment represents where the base application is running and it is what the middleware is going to reflect on and change if needed. In order to use the middleware, the initial configuration should be done creating configuration XMLs, with the specifications about the base system that will be monitored and manipulated, what conditions will generate changes, and what changes will be performed.

4. A Study Case: Middleware for Software Security Improvement

4.1 Problem Statement

In this particular case, the library will be tested only for security aspects; it will make decisions not only about fault recovery, but also to prevent as much as possible security holes when new behavior is adopted by the system. . When an application is built using different software components and these components change dynamically, security aspects have to be considered in order to keep the system secure. For this goal, the middleware must update its information each time that the application is changed because a software

component is removed, exchanged for another one, or updated. There are three general aspects that our middleware must manage when security is studied in a composed system application. They are [4]: *Confidentiality:* It is also known as disclosure; it is the ability of making information available only to authorized users. *Integrity:* This property allows data to be modified only by authorized users. *Authentication:* This is the process of verifying a user's claimed identity. It is the logical step that follows identification. The middleware will allow keeping a system as secure as possible adding these three characteristics in a dynamic way. Each time that the system suffers a change, the library will take care of: - Confidentiality and integrity of the information that is processed. - Authentication of the software components that will be added or upgraded. - Update information related to new, expected risks that software changes can bring to the application.

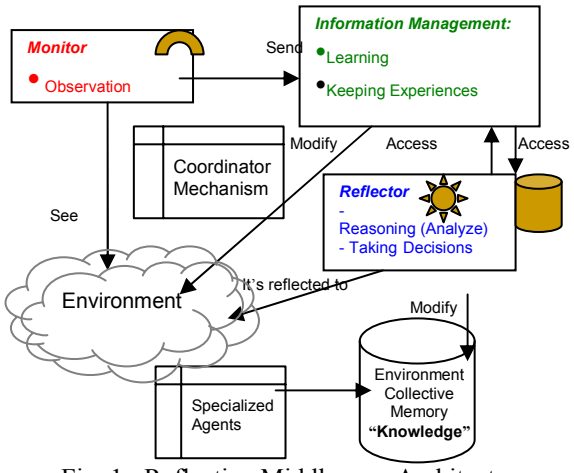

Fig. 1: Reflective Middleware Architecture.

Each software component in a composed application will have a level of security that is related to the way the component uses and manages the information and authentication; this security level is a number from 0 to 3, with 3 denoting a most trusted component (see Table 1). The composed system will be only as secure as its most insecure component. There will be a required security level for the application; if that value decreases below the required level as the result of changes, the middleware must change one or more components in order to increase the value again. A combination of each characteristic with different security levels can be defined for each component of the system, these are called requirements, and the middleware will monitor the behavior of the components and evaluate if they accomplish the requirements.

Security	Confidentiality	Integrity	Authentication
Level			
0	Use of no encryption.	Provide basic process to assure the integrity of information.	Public information.
1	Some encryption needed – No specific encryption method.	The integrity of the information is needed. No verification processes are required to validate the information.	Information and system access are related to each user.
2	Encryption is needed with the strongest available method.	A good validation is required to assure the integrity of the data. The system provides audit mechanisms.	Information and system access are related to each user and a log of all user activities is stored.

Table 1: Secure levels description related to each secure characteristic

4.2 Implementation

This middleware reflects over the complete system software architecture allowing dynamic selection of new software components based on security aspects. The main questions are: how to bring a new component into a "secure" system without losing that security, how to determine the trust of the new component, how to integrate that component, how to select the safest component, and how to change the actual component in order to improve the system security. The tasks that the middleware performs are the following: Monitor: It will check if each component in the system has the required security characteristic; if it realizes that this is not the case, a message is sent to the Reflector. The communication in this case is made through direct messages, the track of the found situation is stored in a XML file which can be accessed by any of the agents in order to look for relevant legacy information that can be used to process information and make the right decision when it is needed. Reflector: It will make the decision as to what action to take. At present, there are only three main actions: change the component, upgrade (Install a new version) the component or tune the existing component (if a tuning is needed for this particular use case, the encryption mechanism is changed). The communication in this case is indirect, and the "experience" about how good the change was is stored in an XML file as historic information. Information Manager: It will store all the information generated after each change. Also, it will give information to the Monitor so it can make the decisions based on the historic events. For this particular case, the most important fact is what encryption mechanism has had less attacks, and which one has kept the

application information safer for more time, this will give to the reflector the idea of what mechanism to choose when a high security level is required by the base application. As this implementation will focus on security, the middleware was configured to check confidentiality, integrity, and authentication of a specific application (Fig 2). In our case, we suppose an application that consists of three software components that work together, each one of the components has different security requirement, some can be updated, and some must only be replaced if fails. For this case, we are focusing the security in the access to the information that the three base components manage. The middleware takes all the initial requirements from a XML file (Fig. 3). Beside Monitor, Reflector and Information Manager, a specialized agent was created to change cryptographic algorithms when is needed, all legacy information for indirect communication is stored in XML files, and the Monitor will check the base system components in order to determine if security was broken. Reflector will make the changes to the base system components (Fig. 2).

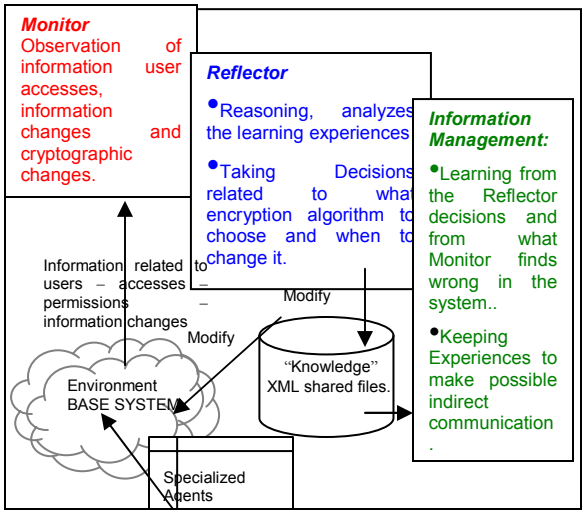

Fig. 2: Testing Architecture (Security Use Case)

4.3. Test Environment

The Middleware Monitor checks the possible security issues in a proactive way. This module was configured to run each ten seconds. The Reflector makes decisions based on both the results that the Monitor gets and the legacy information (in XML files – Fig. 4) stored by the Information Manager. Base system information is updated since the very first time that the middleware start working. The Reflector will get all available information in order to make its final decisions evaluating what is the most successful cryptography mechanisms for that particular system and what system components work in the most secure way. As we can see in Fig. 4 the Management Information stores both, all actions that the user make with the system and the tasks that the middleware performs into the system . This is particularly important for security management, this is part of the legacy information that will be kept in to the XML middleware files. Confidentiality, integrity, and authentication are the aspects that are being evaluated in this particular test. Some main text files that belong to the application should be encrypted, the middleware does it based in the initial requirements. In order to measure each event that the middleware finds, three actions are introduced: *Alarm:* It is a configurable element in the middleware, can be an email that the middleware sends to a group of users, a message on the screen, a line that is written in a log file, or an action that is done as this is configured in the middleware, for example disconnect a user, break network connection or other. *Tuning*: it is the intercession that the middleware makes to the application. In this case the middleware will change the encryption mechanisms that are used by the base system that the middleware is reflecting on. Replacement: When a tuning can not be done due to the own characteristics of the software component, the middleware suggests to the system administrator, a component replacement. This person, who is the responsible of the base system, will make the decision. If an upgrade can be done, the middleware will do it. An upgrade can be done if and only if there is an equal available component with same interfaces, generally a newer version of the same component; otherwise, the programmer team should make the changes. The test application uses four components; each of them was given a set of security requirements and a set of attacks in order to test the middleware. Components had three types of attacks; (i) Change of the information that the base component uses. (ii) Delete component files (iii) And unauthorized access to the information in the base application. The following security characteristics are defined by the base system administrator, and they are part of the initial middleware configuration: *Component A* does not have any security restriction, so neither alarms nor upgrades nor changes were required. *Component B* had some secure requirements, some alarms will be required when it is under attack.. *Component C* had high secure requirements, alarms, tunings, upgrades, and suggestion of changes were done, and the middleware has to maintain a highly secure environment for it. *Component D* had different levels of security, automatically; all levels were set up to the highest security level so the middleware could keep this component under its secure standards.

When two components communicate each other in a common environment, the security level of the weakest

component should rise to the highest security level of the other component in order to keep secure all the system.

```
<Application>
<AppName>Reflective Architecture Test<AppName>
<SoftwareComponent>
        <Name>Geometry</Name>
      <Path>c:/componentloc/geom</Path>
        <Security>
                  <User>BLANCA ABRAHAM</User>
                  <Rights>777</Rights>
                  <Levels>

        <Confidenciality>0</Confidenciality>
                           <Integrity>0</Integrity>

        <Authentication>0</Authentication>
                  </Levels>
        </Security>
        <Performance>
        </Performance>
</SoftwareComponent>
```

Fig. 3: Example of one base system component configuration XML for the Middleware

```
<MIDDLEWARE       CONFIGURATION>03-03-08</MIDDLEWARE
CONFIGURATION>

<Date>08-03-03:16:15:05</Date>
<User>BLANCA ABRAHAM</User>
<Access >1</Access >
<Alarm>1<\Alarm>
<Action>RP<\Action>

<Date>08-03-03:16:20:13</Date>
<User>BLANCA ABRAHAM</User>
<Access>0</Access >
<Alarm>0<\Alarm>
```

Fig. 4 Information Management Files.

In this case, our test application will have security level 2, and configuration of components A and B should change to 2 so the system security can be kept. Then, the middleware needs to assure that the main application will keep the following security aspects during its entire life (Table 2).

Component	Confidentiality Level	Integrity Level	Authentication Level
A	0	0	0
B	1	1	1
C	2	2	2
D	2	1	0

Table 2: Initial request security level description for testing purposes

4.5 Results:

During the test, the alarms generated by the middleware for each component when security issues were found were the following (Table 3):

Component	Number of Alarms related to Confidentiality	Number of Alarms related to Integrity Level	Number of Alarms related to Authentication Level
A	0	0	0
B	3	3	3
C	3	3	3
D	3	3	3

Table 3: Alarms generated by the middleware – All components communicate each other.

The number of alarms means the number of times that the middleware sends a message (email, log file or screen report) to the system administrator, or warning about a weird behavior. In this case, the alarms related to confidentiality, integrity and authentication were sent when no encryption was found in files that should have some, or when the modification date of some encrypted files was modified out of the middleware rules (see Fig 5). In this case, because all components communicate each other, if the security of one is lower the acceptable level, the other components are under risk too, because they all can be accessed using any other component of the system. Because components A and D do not communicate with components B and C, when B and C have security problems, the Middleware only sends an alarm related to them. In this case, if component D is under security levels, alarms for the rest of components are not required because it does not compromise their securities.

When components communicate each other, their dependencies are monitored by the middleware, and the security levels will increase to the most secure component security level. Table 5 shows results when simulation of confidentiality, Integrity and Authentication problems were done for each component, the simulation was created changing confidential information, access of no authorized users and information modification. In this particular implementation, the tuning is related to the encryption mechanism. If an alarm is activated, the encryption mechanism is automatically changed by the middleware (intercession) in order to keep the information protected, if a non authorized user is in, it would be logged off automatically and the network connection can be break too. The software components can have available new versions that are used to update them when it is necessary, that is when the component is not working properly. The upgrades that the middleware made per each component were (Table. 6), where it is seen that only component C had available new versions and it was updated when its security level decreases under the acceptable value. If a new version is not available of a software component when we need to update it because the component is not working properly, a change of component is required which should be done by the system administrator. The number of replacements suggested by the middleware per component are in Table 7:

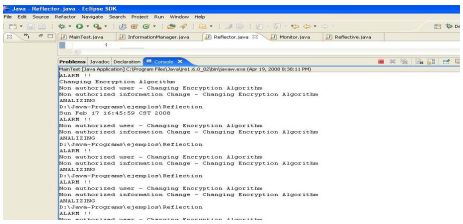

Fig.5: Middleware monitoring and generating alarms because of unusual behavior in the application.

Component	Number of Alarms related to Confidentiality	Number of Alarms related to Integrity Level	Number of Alarms related to Authentication Level
A	0	0	0
B	2	3	2
C	2	3	2
D	1	0	1

Table 4: Alarms generated by the middleware – In this case no all components communicate each other.

Component	Number of Tunings related to Confidentiality	Number of Tunings related to Integrity Level	Number of Tunings related to Authentication Level
A	0	0	0
B	1	3	2
C	2	2	1
D	3	1	3

Table 5: Tunings that were done for each component

Component	Number of Alarms related to Confidentiality	Number of Alarms related to Integrity Level	Number of Alarms related to Authentication Level
A	0	0	0
B	0	0	0
C	1	1	1
D	1	1	1

Table.6: Upgrades made for each component

As can be seen in Table 6 and 7, even when a component B replacement was suggested it could not be done automatically because a new version was not found, for components C and D the update was done.

The change of component should be done by the system administrator, because this task implies extra configuration or programming in order to make the new component work well in the environment. The component selection is done using an algorithm developed in [25], Fig. 6

The implementation of tuning was done by changing or incorporating different encryption methods. The decision of what method to use was made based on the information management knowledge by the middleware.

Component	Number of Replacements	Number of Replacements	Number of Replacements
	related to Confidentiality	related to Integrity Level	related to Authentication Level
A	0	0	0
B	1	1	0
C	1	1	1
D	1	1	1

Table. 7: Suggested replacements for each component.

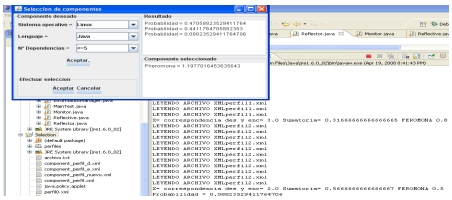

Fig.6: Middleware looking for new components that can replace the application components if needed

5. Conclusions

This middleware not only audits automatically the security of a system that can be composed by several software components, but also make changes, related to component replacement, component tunings and component updates to those components that does not pass the security test, this is particularly important for big systems that have many software components, some of them developed as black boxes. This middleware needs only an initial configuration that will be the base for monitoring the secure levels. This middleware has many advantages, among them are:

- It can be used for any type of application.
- It works for a given base application over any Operating System.
- It is very easy to configure using only XML files
- It has a knowledge base that can be used for other similar systems.
- It is general enough to be implemented in different types of applications.
- It is flexible so specialized agents can be implemented for special needs.

Even when this test was based only for security aspects in an application, we are working to implement it in monitoring performance.

Other researches can be focused in resource management, web services or others activities, this can be done because the middleware can be configured to monitor, change and warn about any process, area or resource that can be checked in running time. Many other interested works can be derived from this initial approach, they are:

- Grids and how their resources are located and load balanced.

- Data warehouses and their security, memory and hard disk management which are some of the main concerns for this type of systems
- Internet services and how to allocate faster and more efficient services,
- Auto-organized systems
- Knowledge Bases that improve the learning mechanisms of the Middleware. Among others.

6. Acknowledge

This work was realized with the support of CDCHT-ULA (Grant I-820-05-02-AA), and FONACIT (Grant 2005000170).

7. References

[1] D. Caromel and J. Vayssiere Reflections on MOPs, Components, and Java Security. European Conference on Object-Oriented Programming, Lectures Notes in Computer Science (LNCS) Budapest, Hungary, June 18-22, pp 256-274, Springer Verlag No 2072,.

[2] M. Adorni, S. Bandini, et al. Model Requirements: Architectural Model, Functional Model, Context Model, Metamode. MAIS Technical Report R1.3.1.,May 2003

3] M. Parashar, S. Harir. Autonomic Computing, Concepts, Infrastructure and Applications. CRC Press, 2007 by Taylor and Francis Group.

[4] F. Stajano. Security for Ubiquitous Computing. Wiley Series in Communications Networking & Distributed Systems. 2002.

[5]http://www.03.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf

[6] W. Cazzola, et al. Architectural Reflection: Concepts, Designs and Evaluation. 1999.

[7] I. Forman, N. Forman. Java Reflection in Action. Manning 2005.

[8]S. Russell, P. Norvig, Artificial Intelligence, A modern Approach. Prentice Hall International. 1995.

[9] P. Grace, et al. GRIDKIT: Pluggable Overlay Networks for Grid Computing. Symposium on Distributed Objects and Applications (DOA), Cyprus, 2004

[10]Objectweb.FRACTALProject.fractal.objectweb.org/ 2008

[11] E. Bruneton, T. Coupave, JB Stefani. Recursive and Dynamic Software Composition with Sharing.

[12] P. Rogers and A. J. Wellings. OpenAda: A Metaobject Protocol for Ada 95. Department of Computer Science. University of York, York, YO1 5DD, UK. 1995

[13] M. Tatsubori, S. Chiba, M Killijian, and K. Itano. OpenJava: A Class-based Macro System for Java. Doctoral Program in Engineering, University of Tsukuba, Tennohdai

[14] J. Itoh and Y. Yokote, "Concurrent Object-Oriented Device Driver Programming in Apertos Operating System", Sony Computer Science Laboratories Technical Memo SCSL-TM-94-005, June 1994.

[15] S. Rusell & P. Norvig, Peter, Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, NJ: Prentice Hall 2003.

[16] Y. Yokote, "The Apertos Reflective Operating System: The Concept and Its Implementation", OOPSLA'92 Proceedings, ACM, 1992 pp.414--434.

[17] Fractal-Programming-Manual Final Release March 10, 2004 INRIA – Nice France.

[18] Open C++. Web Page. http://opencxx.sourceforge.net/ 2008.

[19]Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, K. Kubota. Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach – Real World Computing Partnership. Proceedings of Reflection'96, pp. 141-154, 1996.

[20] F. Ortin Soler, J. Cueva Lovelle. Building a Completely Adaptable Reflective System. Campun Llamaquique, C Calvo Sotelo s/n. 33007 Oviedo Spain.

[21] H. Masuhara, A. Yonezawa. An Object-Oriented Concurrent Reflective Language ABCL/R3 Its Meta-level Design and Efficient Implementation Techniques. Department of Graphics and Computer Science, Graduate School of Arts and Sciences, University of Tokyo Tokyo, 153-8902 Department of Information Science, University of JAPAN.

[22] Proactive Web Page: http://proactive.inria.fr/

[23] Smalltalk: a Reflective Language Fred Rivard Object Technology International Inc. Ottawa – Ontario.

[24] P. Sewell and J. Vitek. Secure composition of insecure components. In Proceedings of the Computer Security Foundations Workshop, CSFW-12, 1999.

[25] B. Abraham. J. Aguilar. J. Batista. "Selection Algorithm Using Artificial Ant Colonies", WSEA Transactions on Computers, Vol. 5, No. 10, 2006. pp. 2197-2203,

[26] P. Herrmann: Formal Security Policy Verification of Distributed Component-Structured Software. In: Springer-Verlag, Berlin, 2003, pages 257-272.

[28] M. Roman, F. Ubicore and R. Campbell. Reflective Middleware: From Your Desk to Your Hand. IEEE Distributed Systems Online. Vol. 2, No. 5, 2001.