Diagnosis Service for Embedded Software Component based Systems

BUI Thi Quynh
LCIS - INPGrenoble
50, rue B. de Laffemas, BP 54
26902 Valence Cedex 9 - France
(33) (0) 4.75.75.94.46
Thi-Quynh.Bui@esisar.inpg.fr

AKTOUF Oum-El-Kheir
LCIS - INPGrenoble
50, rue B. de Laffemas, BP 54
26902 Valence Cedex 9 - France
(33) (0) 4.75.75.94.46
Oum-El-Kheir.Aktouf@esisar.inpg.fr

ABSTRACT

This paper studies the fault diagnosis of component-based applications, especially embedded ones. The principle of the proposed diagnosis technique is to implement inter-component tests in order to detect and locate faulty components without component replication. A diagnosis service for embedded software component based systems is developed. Its advantages are application autonomy, cost-effectiveness and better usage of system resources. Such advantages are very important for embedded systems.

Keywords

Dependability, diagnosis, embedded systems, fault tolerance.

1. INTRODUCTION

Component-based software development paradigm enables the construction of applications by assembling existing self-contained components with well-defined interfaces. The cost of the software development process can then be sharply reduced. In addition, the use of replaceable software components simplifies the implementation and the maintenance of complex applications. Component-based software development models, such as Enterprise JavaBeans [18], Microsoft .Net [14], OSGI [16] and the CORBA Component Model [5], are being widely used and have shown improvements in the software development and maintenance process. Current software systems are becoming more distributed and operate in highly dynamic environments. Therefore, dependability of component-based applications is an important research issue.

In this context, most of the proposed approaches are based on component replication and fault masking. Thus, results are guaranteed to be correct though some faults may corrupt the functioning of some application components. But such solutions are very costly, especially in case of embedded applications with limited resources. An alternative cost-effective solution is system diagnosis that concerns the ability of fault-free components to determine the fault-state of the whole application. It seems more interesting to make embedded distributed applications autonomous with regards to the fault-tolerance problem. That explains our choice to investigate diagnosis-based solutions.

Leading projects on real-time embedded systems have essentially focused on meeting QoS aspects related to timeliness by integrating specific mechanisms into standard-based middlewares, such as CORBA. The fault-tolerance approaches of these projects are based on component replication and fault masking. Examples of such projects are the DECOS [11], the CLEOPATRE [4], the ARCAD [13], the iCMG [10], and the AFT-CCM [7] [8] projects.

The Dependable Embedded Component and System (DECOS) project develops an architecture-based design methodology in order to significantly reduce the design, deployment and life cycle cost of dependable embedded applications in many application domains. In this project, fault-tolerance is implemented by the replication technique within an autonomous fault-tolerance layer integrated in the system. The CLEOPATRE (Composants Logiciels sur Etagères Ouverts Pour les Applications Temps-Réel Embarquées) project develops a library of components for temporal fault management in embedded real-time applications. The ARCAD (Architecture Répartie extensible pour Composants ADaptables) project investigates the integration of a replication service in a component-based infrastructure. It is based on the CORBA Component Model and considers replication as a configurable non-functional aspect in a component-based system. This approach uses interception objects that are responsible for capturing the invocations made to a component in order to trigger necessary actions for replication management. The iCMG project is a server-side infrastructure for development, assembly, deployment and management of CORBA Components. The fault-tolerance mechanism is integrated into the component server for fault detection and system recovery. The Adaptive Fault-Tolerance model in the CORBA Component Model (AFT-CCM) is formed by software components that are responsible for implementing fault-tolerance techniques, defining and controlling the behavior of a replicated service.

All these replication techniques are costly and resource consuming, and more efficient solutions should be proposed. In our work, dependability of such component-based applications is studied from the diagnosis point of view. However, we should note that it is not obvious to compare the costs of the replication and the system diagnosis approaches because of their differences and their unknowns. Indeed, while the objective of replication is fault masking, system diagnosis aims to detect and locate faulty

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EFTS'07, September 4, 2007, Dubrovnik, Croatia
Copyright 2007 ACM ISBN 978-1-59593-725-4/07/09...$5.00

components within a system. Once the diagnosis process is completed, system is able to isolate them, ignore their output and initiate reconfiguration operation such that the reliability of the system can be maintained in the long run.

As a first step in our work, this paper presents a diagnosis approach based on inter-component testing. It is expected that this approach should enhance application dependability with a competitive cost-performance trade-off. To support this statement, we conduct two experimental case studies, in a banking and an air conditioning systems. The obtained results constitute premises for our goal of building fault tolerant applications in embedded systems.

This paper is organized as follows: Section 2 introduces a global view of the proposed diagnosis approach. Section 3 investigates the description of the diagnosis service. The obtained experimental results are presented in Section 4. Finally, Section 5 gives some concluding remarks.

2. GLOBAL DIAGNOSIS APPROACH

System-level diagnosis is a process initially introduced by Preparata *et al.* [17] that allows to determine the fault-state of distributed system components. Its basic idea is to use inter-component tests, *i.e.* a component tests another one to determine its fault state. System-level diagnosis may rely on centralized diagnosis or distributed diagnosis. In the former, it is assumed that a central reliable device monitors test execution and computes the system state on the basis of test results. In the latter, it is supposed that there is no central device and each fault-free component is aware of the whole system state. So, such algorithms they are called self-diagnosis algorithms. Depending on the manner in which tests are assigned, diagnosis algorithms can be static or adaptive. In static algorithms, test relations are established before the diagnosis process. On the contrary, they are dynamic in the adaptive strategy and depend on previous test results that indicate the state of some system components.

Our proposed approach for diagnosing faulty components consists of two main aspects. The first one concerns the execution of on-line inter-component tests, *i.e.* testing a component to serve the diagnosis process during the application execution that requires the integration of the test functionality within a component. The second one is the diagnosis process itself that consists of analyzing inter-component test results for determining the fault state of the whole system. Several diagnosis strategies have been proposed [2][12][17]. These diagnosis strategies ensure a deep knowledge of the state of system components and communication links between them.

The basic idea of the proposed diagnosis approach is to partition the application into diagnosis groups where inter-component tests are performed following given test assignments. The partitioning approaches perform better than non-partitioning ones that cover the whole system by reducing unnecessary extra message traffic and processing time.

The main aspects that should be studied in an on-line inter-component testing concern the test code and test data demands regarding system resources, i.e. memory and CPU time. For CPU scheduling, the basic idea is to use component's idle cycles to perform on-line testing, such as in [6]. The problem we are investigating consists of proposing such an approach in a software component-based application. The memory usage logically depends on test precision degree. As deployed components have intensively been tested as stand-alone components before being integrated into a global application, lightweight on-line tests, with minimum memory occupation, should be sufficient. This is taken into account in our approach described in the following.

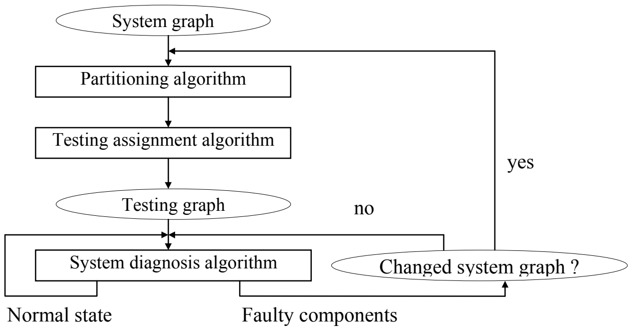

Figure 1. Adaptive system diagnosis procedure with a partitioning approach

Figure 1 represents an adaptive system diagnosis procedure with a partitioning approach. A partitioning algorithm divides the original system graph into smaller groups. Then, testing assignment algorithm, that produces a testing graph from the system graph, is performed within each group. After the diagnosis process, if the system configuration changes, then a new testing graph is obtained by partitioning and testing assignment algorithms within the related groups.

A diagnosis group is defined as a group of components that use the same diagnosis model and the same diagnosis algorithm. For example, in Figure 2, the components of the Group 1 and Group 2 execute diagnosis algorithms 1 and 2, respectively.

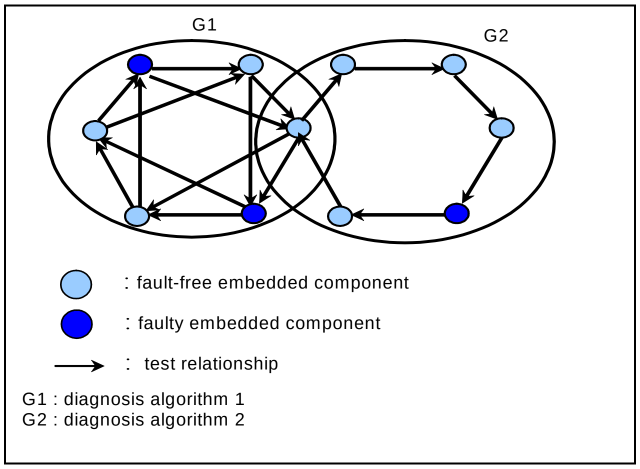
G1
G2
: fault-free embedded component
: faulty embedded component
: test relationship
G1 : diagnosis algorithm 1
G2 : diagnosis algorithm 2

Figure 2. Diagnosis groups

3. DIAGNOSIS SERVICE DESCRIPTION

The architecture of the proposed diagnosis service is illustrated in Figure 3. This diagnosis service provides three types of interfaces: interfaces of the member component side (interfaces of the tested component side and interfaces of the tester component side), interfaces of service component side and interfaces of observer side. These interfaces are extended from the system components except the interfaces of the service component side, as the service component is designed for the diagnosis service and is a standalone component.

- The interfaces of the member component side provide inter-testing component facilities, inform service component of all available test cases and send the diagnosis result to the observer.
- The interfaces of the service component side manage the diagnosis groups. They provide member components of a diagnosis group with intelligent testing strategies.
- The interfaces of the observer side are provided to an external component that aims to know the fault state of the diagnosis group or of the whole system.

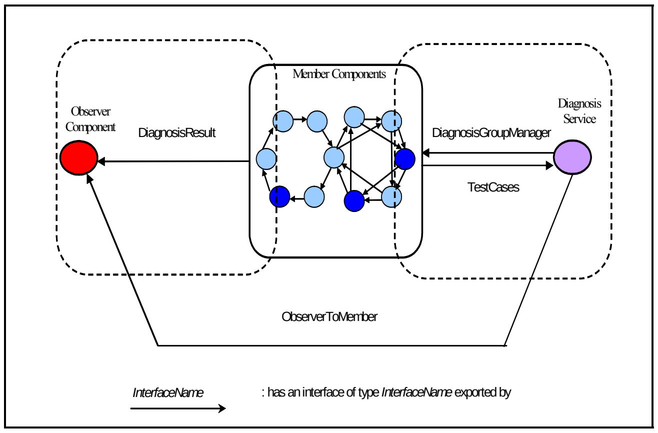

Figure 3. The architecture of the proposed diagnosis service

We describe below these interfaces (Sections 3.1, 3.2 and 3.3) and their interactions in Section 3.4.

3.1 Interfaces of the Member Component

3.1.1 Interfaces of the Tested Component Side

These interfaces are integrated into the component in the form of a test interface that provides testing facilities and informs service component of all available test cases.

A component contains a set of provided and required interfaces. Each provided interface is a set of operations that the component provides to other components. While each required interface is a set of operations that the component needs to perform its execution. In a similar manner, testing facilities are another service that the component provides to its environment. As all other services, test facilities are provided through a number of interfaces: in this case the test interface (Figure 4).

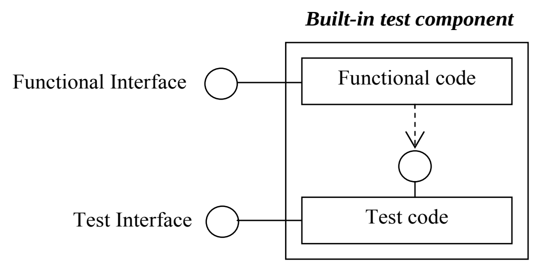

Figure 4. Built-in test component

A component can be generally viewed as a state machine and requires state-transition testing. Before the execution of a test, the tested component must be brought into the initial state required for a particular test. After test-case execution, the test must verify that the outcome (if generated) is as expected, and that the tested component resides in the expected final state. To reach this goal, we use an additional testing interface that contains special purpose operations for setting and retrieving the internal state of a component [9], as shown in Figure 5.

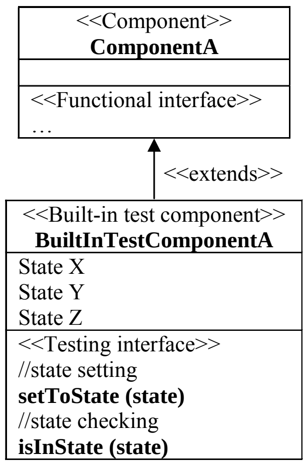

Figure 5. Concepts of built-in test component and testing interface

A testing interface extends the normal functionality of the component. It is implemented as a component extension in its own right, so that the implementation of the testing software is encapsulated and strictly separated from the normal functional software. A testing interface comprises operations for setting and getting internal state information which are *setToState* and *isInState.*

The state checking operation (*isInState(state)*) of the testing interface verifies whether the component is currently residing in a distinct logical state. The state setting operation (*setToState*) sets the component's internal attributes to represent a distinct logical state.

More details on the testing interfaces are presented in [3].

The tested component is described in Figure 6.

```
//IDL
Component testedComponent extends component
{
        Provides TestInterface
        Provides TestCases
}
```

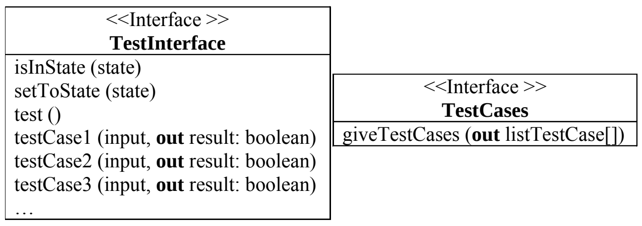
<<Interface >>
TestInterface
isInState (state)
setToState (state)
test ()
testCase1 (input, out result: boolean)
testCase2 (input, out result: boolean)
testCase3 (input, out result: boolean)
…
<<Interface >>
TestCases
giveTestCases (out listTestCase[])

Figure 6. Interfaces of tested component side

3.1.2 *Interfaces of the Tester Component Side*

As in 3.1.1, the test interface of the tested component provides component testing facilities. The interface of the tester component requires it (Figure 7) to test the functionality of the tested component with the test cases indicated by the service component.

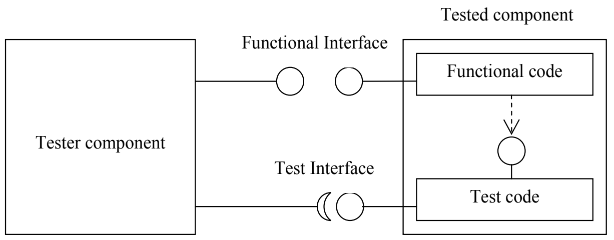

Figure 7. Tester component uses test interface of tested component

The tester component also uses the interface *DiagnosisGroupManager* of the service component to know the test strategy of its group in order to trigger the tests and send diagnosis information to other components.

If the diagnosis algorithm is distributed, or if this component is the central component of a centralized diagnosis, it gets diagnosis information of the other components, analyzes test results (called the syndrome) and provides the global state of a given group or of whole system to the observer using the interface *DiagnosisResult*.

The tester component is defined in Figure 8:

```
//IDL
component testerComponent extends component
{
      uses TestInterface
      uses DiagnosisGroupManager
      provides DiagnosisInformation
      provides DiagnosisResult
      uses GetDiagnosisInformation
      }
```

<<interface>> DiagnosisInformation
giveDiagnosisInformation ()

<<interface>> DiagnosisResult
Diagnose (syndrome, **out** SystemState)

Figure 8. Interfaces of tester component side

3.2 Interfaces of the Service Component Side

These interfaces provide member components of a diagnosis group with the intelligent testing strategy (according to diagnosis methods such as, static or adaptive, distributed or centralized diagnosis algorithm). Their principal functionalities are to:

- partition the system in groups,
- determine the test strategy in each group,
- receive the test cases of each component,
- analyze the scheduling information to take benefit from the idle cycles of a component to test it,
- send the testing assignment information to the components.

Figure 9 gives an example of a testing graph of five components. It is a case of a centralized diagnosis in which component 4 is assumed to be the central component.

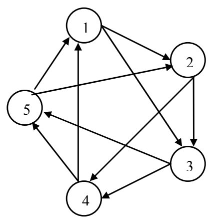

Figure 9. A testing graph example

Testing assignment information is presented in Table 1. This table shows that, for example, component 1 will test component 2 with test cases 1, 2, 4 and sends the test result to component 4.

Table 1. Testing assignment information

Tester component	Tested component	Test cases	Send results to components
1	2 3	1, 2, 4 1,4	4
2	3 4	2,3 3,5	4
3	4 5	3,4 2,4	4
4	5 1	1,4 1,2,5	4
5	1 2	1,4 2,5	4

Moreover, it provides the information about the components to the observer by the interface *ObserverToMember* so that the observer can get the diagnosis result from these components.

The service component is described in Figure 10.

```
//IDL
Component serviceDiagnosis
{
            uses TestCases
            provides DiagnosisGroupManager
            provides ObserverToMember
}
```

<<interface>> DiagnosisGroupManager
Partition (systemGraph, **out** groups) Analyze (groups, **out** testStrategy) MemberToMember(**out** testAssignment)

<<interface>> ObserverToMember
ObserverToMember (**out** MemberDiagnostician)

Figure 10. Interfaces of service component side

3.3 Interfaces of the Observer Side

These interfaces are provided to an external component that aims to know the fault state of the diagnosis group or the fault state of the whole system.

This component uses the interface *ObserverToMember* of the diagnosis service component to know which component makes the diagnosis (by analyzing the syndrome) in each group or in the system. Then the observer side uses the *DiagnosisResult* interface of these components to get the global fault state of the group or of the system.

The interfaces of observer component are described as follow:

```
//IDL
Component Observer extends component
{
        uses ObserverToMember
        uses DiagnosisResult
}
```

3.4 Component Interactions

Interactions between the member components in one group and the diagnosis service component are shown in Figure 11. The service component receives all available test cases of tested components. Then, it analyzes the scheduling information and sends the testing assignment information to tester components by the interface *MemberToMember*. After receiving the testing assignment, a tester component tests tested components and pushes the test result or diagnosis information to components defined by the service component according to the predetermined test strategy.

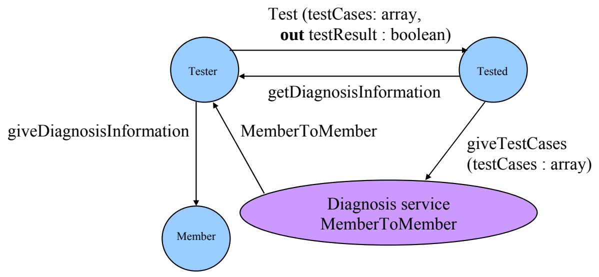

Figure 11. Interactions of member components and service component

Figure 12 shows interactions involving observer component, member components, and service component. The observer uses the interface *ObserverToMember* to know the components in each group or in the whole system. After inter-component testing and syndrome analysis, the observer gets the state of a group or the whole system from these components.

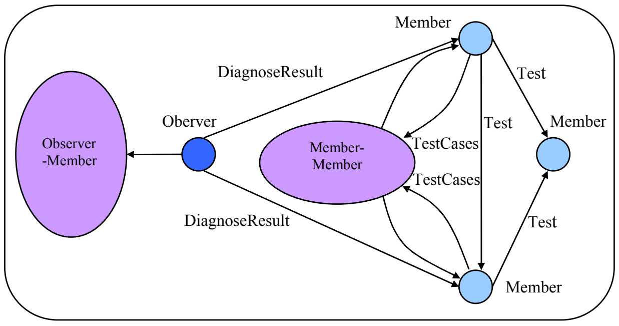

Figure 12. Interactions of member components, observer and service component

4. EXPERIMENTATION

To implement our proposed service, we carry out two case studies: a banking system and an air conditioning system. In these case studies, we adopt two diagnosis algorithms from the literature. The first one is a distributed diagnosis algorithm that makes every component in the system aware of the whole system state [2]. The second one is a centralized algorithm that relies on a central component to determine the fault state of the whole system [17].

These case studies are implemented with the OpenCCM platform [15]. The centralized diagnosis method and the distributed diagnosis method are executed for a comparison purpose.

The obtained results presented in the following aim to give indications for the future orientation of the proposed diagnosis service implementation.

The first results (Table 2) are measured on the banking example that has 4 components: code validation component, deposit component, withdraw component, and consult component.

- Code validation component: verifies the banking card's Pin code provided by the user. The card is locked after three times of providing a false Pin code.
- Deposit component: is in charge of money deposit.
- Withdraw component: withdraws money.
- Consult component: provides the user's detail account.

Table 2. The obtained results of the banking system

Criterion	Distributed diagnosis method	Centralized diagnosis method
Number of components needed	5	6
Fault number (t)	$3 \geq t$	$1 \geq t$
Fault detection time (ms)	13442	12062

The number of needed components consists of four system components, plus one service component (see Figure 3) and one central reliable component, as we are in the centralized diagnosis approach.

To determine the maximum fault number (t), we used the general result presented in [1] which states that for a system with n components:

- for the centralized diagnosis approach, $n \geq 2t+1$,
- for the distributed one, $n \geq t+1$.

The second case study is the air conditioning system example which consists of 4 components:

- Thermometer component: transmits the temperature into the room.
- Thermostat component: is identical to the thermometer component but allows the user to choose his desired temperature.
- Administrator component: manages and controls the thermometer and the thermostat and changes the temperature remotely.
- Temperature engine component: controls the heating valve and the cooling valve to adjust the air supply temperature to the room.

The obtained results of the air conditioning system are shown in Table 3:

Table 3. The obtained results of the air conditioning system

Criterion	Distributed diagnosis method	Centralized diagnosis method
Number of components needed	5	6
Fault number (t)	$3 \geq t$	$1 \geq t$
Fault detection time (ms)	12452	10962

Similar to the first case study, we have, in total, 5 components in the distributed diagnosis method and 6 components in the centralized one. The maximum fault number is also determined in the same formula.

As shown in Table 2 and Table 3, we find that:

- The distributed diagnosis method can tolerate up to 3 faulty components, whereas the centralized method can detect and locate 1 faulty component.
- The fault detection time of the distributed diagnosis method is longer than the centralized diagnosis method one.

The result shows that, they should be a trade-off in resource utilisation between these methods. Distributed method can tolerate more faulty components, yet needs more time to detect fault. Our ongoing work calculates other resource parameters in the system, such that memory, CPU consuming, etc. More complete results will provide indicators to developers that aim to build a fault tolerant system based on the proposed diagnosis approach.

5. CONCLUSION

The presented work, even if in its beginning step, is promising. The proposed inter-component and diagnosis approaches are very interesting functionalities since they may enhance application dependability with a competitive cost-performance trade-off, in comparison with classical costly replication approaches.

Our approach offers a number of possibilities for near future research. It would be interesting to use idle cycles to perform tests and studying efficient partitioning algorithms in order to reduce the fault detection time and improve the impact of diagnosis method on system performance. Another natural extension is continuing on completing a fault tolerance method from the diagnosis point of view. The diagnosis process will be followed by a system reconfiguration, such that the reliability of the system can be maintained in the long run.

6. REFERENCES

[1] Barborak, M., Makek, M. and Dahbura, A., "*The consensus Problem in Fault-Tolerant Computing*", ACM Computing Surveys, Vol.25, No.2, pp. 171-220, June 1993.

[2] Bianchini, R., Buskens, R. W., "*An adaptative distributed system level diagnosis algorithm and its implementation*", Proceedings of the 21st international IEEE Symposium on Fault-Tolerant Computing, IEEE CS Press, pp. 616-626, Montreal, Canada, June 1991.

[3] Bui, T. Q. and Aktouf, O., "*Inter-component testing for system-level diagnosis of embedded component based applications*", Proceedings of the 7th Multidisciplinary International Conference on Quality and Reliability, pp. 246-254, Tanger, Marocco, March 2007.

[4] Cleopatre, available: http://www.cleopatre-project.org.

[5] CORBA Components, OMG Document formal/02-06-65, 2002, available: http://www.omg.org.

[6] Dahbura, A.-T., "*An O($n^{2,5}$) fault identification algorithm for diagnosticable systems*", IEEE Transactions on Computers, vol. C-33, n°6, pp. 486-492, June 1984.

[7] Favarim, F., Fraga, J. and Siqueira, F., "*Fault-tolerant CORBA Components*", Proceedings of the 2nd Workshop on Reflective and Adaptive Middleware, IEEE CS Press, pp. 144-148, Rio de Janeiro, Brazil, June 2003.

[8] Fraga, J., Siqueira, F. and Favarim, F., "*An Adaptive Fault-Tolerant Component Model*", Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, IEEE CS Press, pp. 179-186, Capri Island, Italy, October 2003.

[9] Groß, H. G., "*Built-in Contract Testing in Component-based Application Engineering*", CologNet Joint Workshop on Component-based Software Development and Implementation Technology for Computational Logic, Affiliated with LOPSTR, Madrid, Spain, September 2002.

[10] ICMG, available: http://www.icmgworld.com.

[11] Kopetz, H. and Wien, T., "*DECOS - European Integrated Project Proposal*", available: https://www.decos.at/download/021003-DECOS.Grenoble-US.pdf/, October 2002.

[12] Lee, K. S. and Shin, G., "*Probabilistic Diagnosis of Multiprocessor Systems*", ACM Computing Surveys, Vol.26, No.1, pp. 121-129, March 1994.

[13] Marangozova, V. and Hagimont, D., "*An Infrastructure for CORBA Component Replication*", Proceedings of the 1st IFIP/ACM Working Conference on Component Deployment, LNCS 2370, Springer-Verlag, pp. 222-232, Berlin, Germany, June 2002.

[14] Microsoft, "*Overview of the .NET Framework*", MSDN Library White Paper, 2001, available: http://msdn.microsoft.com.

[15] OpenCCM, available: http://www.objectweb.org.

[16] OSGI, available: http://www.osgi.org.

[17] Prerapata, F. P., Metz, G. and Chien, R. T., "On the connection assignment problem of diagnosticable systems", IEEE Transactions on Electronic Computers, vol. EC-16, n°6, pp. 848-854, December 1967.

[18] Sun Microsystems, "*Enterprise JavaBeans Specification*", v2.0, 2001, available: http://java.sun.com/ejb/.