7ième édition du congrès international pluridisciplinaire
Qualita 2007 – Tanger (Maroc)
Du 20 au 22 mars 2007

INTER-COMPONENT TESTING FOR SYSTEM-LEVEL DIAGNOSIS OF EMBEDDED COMPONENT-BASED APPLICATIONS

BUI Thi Quynh – 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, {Thi-Quynh.Bui, Oum-El-Kheir.Aktouf}@esisar.inpg.fr

Résumé :

Ce papier présente la sûreté de fonctionnement des applications développées à base de composants, en particulier les applications embarquées du point de vue du diagnostic. Le principe de la technique du diagnostic est de mettre en place des tests inter-composants afin de détecter et localiser des composants défaillants sans avoir recours à la réplication. L'approche proposée pour diagnostiquer des composants fautifs se compose de deux aspects principaux. Le premier concerne l'exécution des tests inter-composants qui nécessite l'intégration de la fonctionnalité de test dans un composant, qui est aussi l'objectif de ce papier. Le second est le processus de diagnostic lui-même qui concerne l'analyse des résultats des tests pour déterminer l'état global du système. Les avantages de cette méthode de diagnostic comparés aux techniques de tolérance aux fautes par redondance sont l'autonomie de l'application du point de vue de la sûreté de fonctionnement, la réduction des coûts liés à la tolérance aux fautes, et une meilleure utilisation des ressources du système. Ces avantages sont très importants dans beaucoup de systèmes et particulièrement dans les systèmes embarqués.

Abstract:

This paper studies the dependability of component-based applications, especially embedded ones, from the diagnosis point of view. The principle of the diagnosis technique is to implement inter-component tests in order to detect and locate the faulty components without redundancy. The proposed approach for diagnosing faulty components consists of two main aspects. The first one concerns the execution of the inter-component tests which requires integrating test functionality within a component, and which is also the objective of this paper. The second one is the diagnosis process itself which consists of the analysis of inter-component test results to determine the fault-state of the whole system. Advantages of this diagnosis method when compared to classical redundancy fault-tolerant techniques are application autonomy, cost-effectiveness and better usage of system resources. Such advantages are very important for many systems and especially for embedded ones.

Mots clés : Sûreté de fonctionnement, diagnostic, bus logiciels, systèmes embarqués, tolérance aux fautes, tests inter-composants.

Keywords : Dependability, diagnosis, middlewares, embedded systems, fault tolerance, inter-component testing.

1. Introduction

The component-based software development approach has been pointed out as a new milestone in the history of software development. By composing applications from existing self-contained components with well defined interfaces, the cost of the software development process can be reduced sharply. In addition, the use of replaceable software components simplifies the implementation and the maintenance of complex applications. Commercial component-based software development models, such as Enterprise JavaBeans [Sun], Microsoft .Net [Microsoft], and the CORBA Component Model [Omg], are being used widely and have shown improvements in the software development and maintenance process. Current

software systems are becoming even more distributed and operating in highly dynamic environments. Thus, 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 alternate cost-effective solution is system diagnosis that concerns the ability of fault-free components to determine the fault-state of the whole application. This seems more interesting for making embedded distributed applications autonomous with regards to the fault-tolerance problem, this is why we chose to investigate diagnosis-based solutions.

Leading projects in the field of 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 theses projects are based on component replication and fault masking. Examples of such projects are the DECOS [Kopetz], the CLEOPATRE [Cleopatre], the ARCAD [Marangozova], the iCMG [Icmg], and the AFT-CCM [Favarim] [Fraga] 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 the temporal faults management of 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 allows that replication can be managed as a configurable non-functional aspect in a component-based system. This approach uses interception objects that are responsible of 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. Finally, 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.

However, all these replication techniques are very costly and resource consuming, thus more efficient solutions should be proposed. In this work, dependability of such component-based applications is studied from the diagnosis point of view. A diagnosis approach based on inter-component testing is presented. It is expected that this approach should enhance application dependability with a competitive cost-performance trade-off.

This paper is organized as follows: Section 2 gives a global view of the proposed diagnosis approach. Section 3 investigates inter-component tests and describes how to integrate test functionality into a component. The obtained experimental results are given in section 4. Section 5 gives some concluding remarks.

2. The global diagnosis approach

The proposed approach for diagnosing faulty components consists of two main aspects. The first one concerns the execution of the inter-component tests which requires the integration of the test functionality within a component, and the second one is the diagnosis process itself which consists of analyzing inter-component test results for determining the fault state of the whole system. Several diagnosis strategies have been proposed [Barborak][Lee]. These diagnosis strategies ensure a deep knowledge of the state of system components and communication links between them. Very good reviews have presented the main models and strategies proposed for systems diagnosis.

The basic idea of proposed diagnosis approach is to partition the application into diagnosis groups where inter-component tests are performed following given test assignments [Aktouf]. Then, results are transmitted hierarchically to a central observer, so that it is possible to analyse test results and to determine the fault state (correct, faulty) of each component.

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 1 the components of group 1 and 2 execute the diagnosis algorithm 1 and 2, respectively.

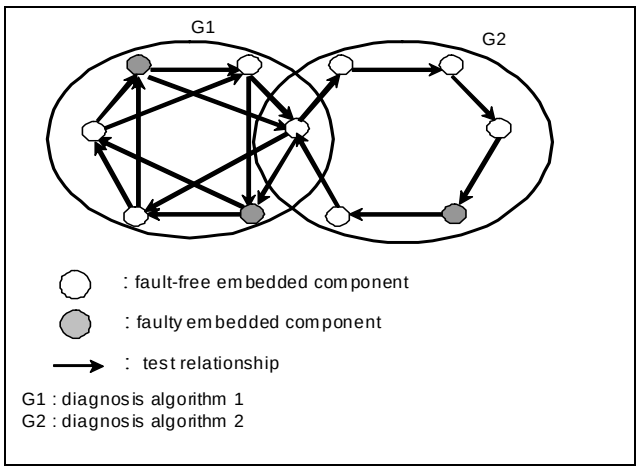

Figure 1. *Diagnosis groups*

The proposed implementation of the diagnosis approach consists of a diagnosis service in order to facilitate its integration within a component framework like the CORBA framework. This diagnosis service provides three types of interfaces (see figure 2).

a. Interface of the observer side. This interface is provided to an external component that aims to know the fault state of the diagnosis group.

b. Interface of the member component side. This interface allows a component to join a diagnosis group or to leave it, and to launch the diagnosis process.

c. Interface of the service component side. This interface provides member components of a diagnosis group with intelligent testing and diagnosis capabilities.

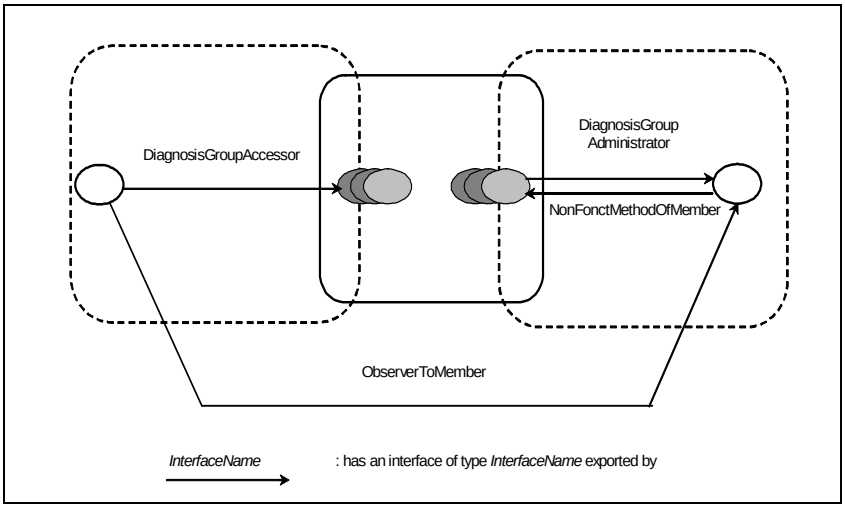

Figure 2. *The architecture of the proposed diagnosis service*

In the following, we will focus on the test functionality and its integration within a component.

3. Inter-component testing

Integrating test functionality within a component corresponds to a built-in test. Such ability has been investigated in previous works, but with the main objective of testing a component within its new execution environment while deploying a component-based application [Atkinson] [Belloir] [Groß] [Wang] [Martins]. This is a good departure point for our research work, but we are mainly interested here in on-line inter-component testing, i.e. testing a component to serve the diagnosis process during the application execution.

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

3.1. Built-in testing interface

A component consists of a set of provided and required interfaces. Each provided interface is a set of operations that the component provides, while each required interface is a set of operations that the component requires in order to perform its operations. In the same way, testing facilities are just 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 built-in test (BIT) interface (figure 3).

A component can generally be viewed as a state machine and requires state-transition testing. Before a test can be executed, 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.

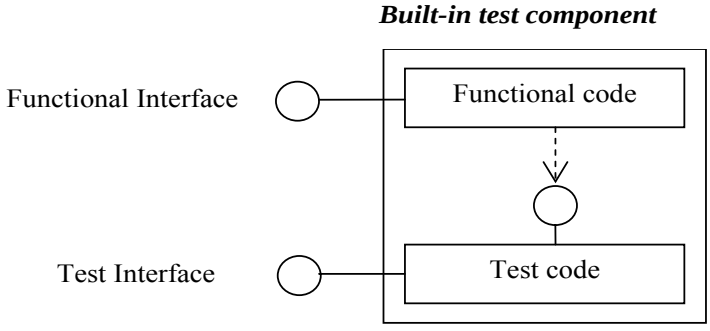

Figure 3. *Built-in test component*

By definition, however, the internal states of a component are hidden to external entities through the principles of information hiding and encapsulation. Therefore, test software cannot usually set or get internal states except through the normal functional interface of the component. A specific sequence of operation invocations through the normal functional interface is usually required to set a distinct state required for a test execution. However, since the tests are performed to verify that the functional interface behaves as expected, it is unwise to use the functional interface to set and verify the internal states of a component, and check the outcome of the tests. In other words, we should not use something for performing a test knowing that it is actually the subject of that test. This problem can be circumvented by using an additional testing interface which contains special purpose operations for setting and retrieving the internal state of a component [Groß] (figure 4).

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.*

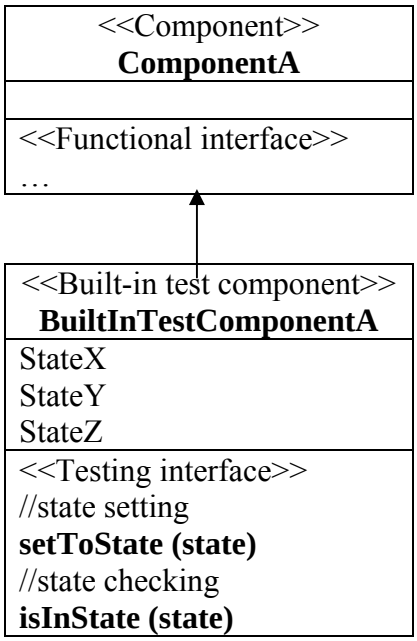

Figure 4. *Concepts of built-in test component and testing interface*

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(state)*) sets the component's internal attributes to represent a distinct logical state.

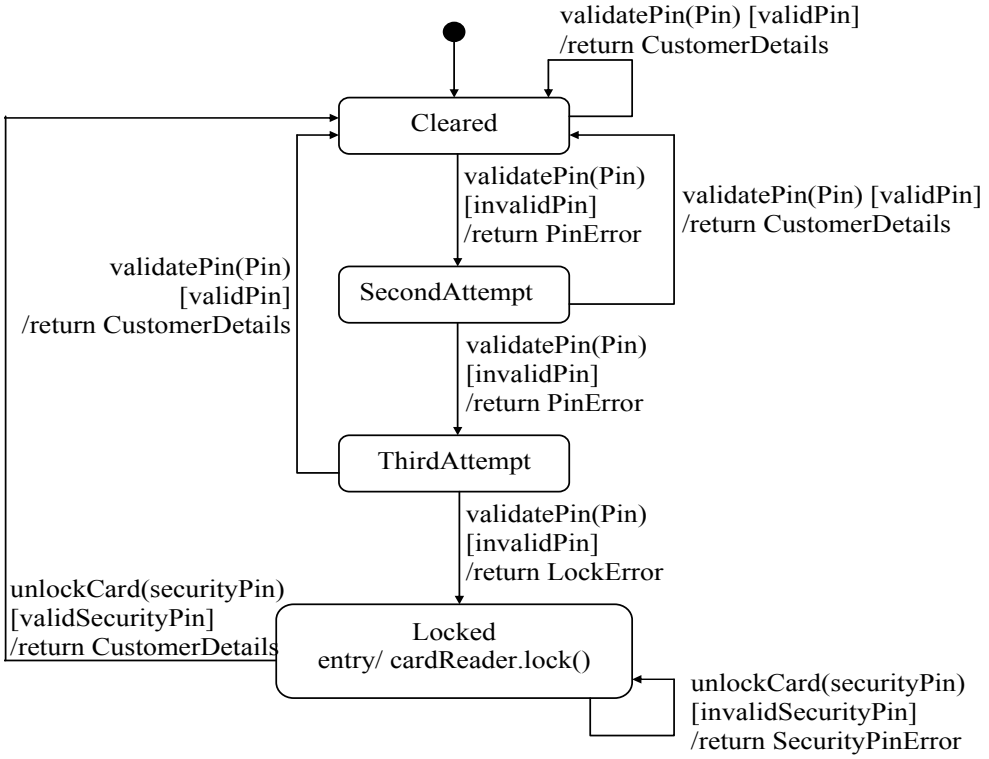

Figure 5. *State model of a banking card [Groß]*

Let us consider the example of the security mechanism of a banking card (figure 5). The card will be locked after a limited maximal number of attempts of providing the wrong pin code. The attribute that determines this security behavior is the number of unsuccessful attempts, an internal attribute of that component. Typically, the card will be locked after three unsuccessful attempts of providing a pin. This count will always be set to zero (*state cleared*) if the correct pin has been provided. This maps to the

state model depicted in figure 5. The defined states are "*cleared*", indicating successful attempt, and "*locked*" in order to indicate that the banking card will be retained by the teller machine. This state model represents only an abstract view limited to the security mechanism of a banking card's total state machine.

A functional or black box test must verify that the state transitions during operation comply with the specification of the tested component. Each identified transition in the state model must be tested. The guard conditions in the state model define alternative transitions that are executed according to distinct input parameters or attribute values.

Figure 6 shows the structural model of an extended banking card component with testing interfaces. Each state is defined as public attribute, and two parameterized operations *setToState* and *isInState* that take these attributes as input for respectively setting the state, and checking whether the component is residing in a given state.

The testing interface for each tested component will be specified according to the realization of the functionality of that component.

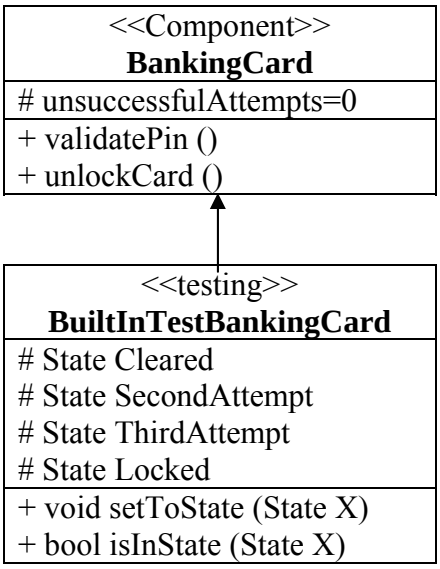

Figure 6. *Structural model of the banking card testing interface*

The banking card example exhibits four different states that represent a counter of the number of unsuccessful attempts of providing the correct banking card pin. So, the implementation of the state setup and checking operations is straightforward. The realization of the testing interface is represented by the activity diagrams in figure 7.

BuiltInTestBankingCard::setToState

case: **State** in
Cleared:
unsuccessfulAttempts = 0
SecondAttempt:
unsuccessfulAttempts = 1
ThirdAttempt:
unsuccessfulAttempts = 2
Locked:
unsuccessfulAttempts = 3
default: continue

BuiltInTestBankingCard::isInState

case: **State** in
Cleared: return
(unsuccessfulAttempts = = 0)
SecondAttempt: return
(unsuccessfulAttempts = = 1)
ThirdAttempt: return
(unsuccessfulAttempts = = 2)
Locked: return
(unsuccessfulAttempts = = 3)
default: return false

Figure 7. *Realization of a testing interface*

3.2. Test cases

In the initial approach of built-in testing (BIT) as proposed by Wang et al. [Wang], complete test cases are put inside the components and are therefore automatically reused with the component. While this strategy seems attractive at first sight, it is not flexible enough to suit the general case. Because the purpose of that built-in testing is testing the component in a new environment, a component needs different types of tests in different environments and it is neither feasible nor flexible to have them all built in permanently. To solve this problem, under the testing paradigm of component + [Groß], test cases are separated from their respective components and put in separate tester components. Another approach to BIT has been proposed by Martins et al. [Martins]. They put a minimal number of tests, like assertions, inside the components, which are reused together with a test specification. However, specific software has to be used in order to transform the test specification into real tests.

In our work, we chose to put the test suite inside the components. Indeed, as described earlier, the goal of our tests is to serve the diagnosis process, where components test each other. So, putting test cases in tester components would be costly. Moreover, test cases have to be lightweight and efficient, for example, generating high efficiency test cases, focused on boundary testing, etc.

```
//Test Case 1: test the third attempt with an error pin
TestCase1()
// Store persistent data of the system
…
// Put the TestableComponent in a specific state before the test
setToState (“ThirdAttempt”);
// Execution of the test operation “validatePin(pin)”
validatePin(pin)
//check the result of operation and the final state
 if (validPin= =false)
        if (isInstate (“locked”))
        testResult=”OK”
        else
        testResult=”FALSE”
else
testResult=”FALSE”
// restore persistent data of the system
…
//Test Case 2:
….
//Test Case 3:
…..
```

Figure 8*. Example of test cases*

Figure 8 shows the example of the test case that tests the third attempt of providing a wrong pin code. First of all, we have to bring the banking card component to the state “*ThirdAttempt*” and we execute the operation “*validate(pin)*” to get the wrong pin code from the user. Then we check the result of that operation and the final state of the banking card to determine whether the test fails or not.

On-line testing is carried out in the real environment, so, when testing ends, if the testing operations have modified some persistent data in the system, those data should be restored to their previous value, to keep the correctness of system states.

As described earlier in section 3.1, the testing interface is one of the provided interfaces of the component, so the tester component can use this interface in the same way as the other functional interfaces.

4. System-level diagnosis versus component replication

It is difficult to compare the costs of the redundancy and the system diagnosis because of their differences and their unknowns. Indeed, while the objective of system diagnosis is to detect and locate faulty components within a system, redundancy aims at masking them.

Because of these major differences, research in these two fields evolves separately and a comparison of the obtained results in each field is difficult to make. In their proposed review of these two approaches, Barborak et al. [Barborak] five a deep qualitative comparison of system diagnosis and redundancy with respect to several criteria (objective, fault assumption, implementation, etc.).

In our work, we conducted an experimental evaluation on the banking example, using two diagnosis algorithms of the literature. The first one is a distributed diagnosis algorithm that makes every component in the system aware of the whole system state [Bianchini]. The second one is a centralized algorithm that relies on a central component to determine the fault state of the whole system [Prerapata].

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

The results are measured on the banking card example which consists of 4 components and is implemented with the OpenCCM platform [Openccm]. The redundancy method, the centralized diagnosis method and the distributed diagnosis method are executed for a comparison purpose.

As shown in table 1, we find that:

- The diagnosis methods use very less resources than the replication method.
- The Adaptive DSD diagnosis method can tolerate up to 3 faulty components, the One-step t-diagnosable method can detect and locate 1 faulty component and the redundancy method can mask until 4 faulty components.
- The fault detection time of the distributed diagnosis method is longer than the centralized diagnosis method, but the fault detection time of the replication method is undefined as there is no fault detection.

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

- for the centralized diagnosis approach, $n \geq 2t+1$,
- for the distributed one, $n \geq t+1$,
- and for the redundancy approach, $n \geq 3t+1$.

Faults are masked with redundancy and detected with diagnosis approach.

Criterion	Distributed diagnosis method	Centralized diagnosis method	Redundancy method
Resource needs	5 components	6 components	13 components
Fault number (t)	$4 \geq t + 1$	$4 \geq 2t + 1$	$13 \geq 3t + 1$
Fault detection time (ms)	13442	12062	undefined

Table 1. The obtained experimental results

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 redundancy approaches.

Références :

[Aktouf] Aktouf, Gacemi. *Component Diagnosis for Distributed Applications*. Qualita Symposium, Nancy, France, 2003.

[Atkinson] Atkinson, Groß. *Built-in contract testing in model-driven, component-based development*. In ICSR-7 Workshop on Component-Based Development Processes, Austin, Texas, 2002.

[Barborak] Barborak, Makek, Dahbura. *The consensus Problem in Fault-Tolerant Computing.* ACM Computing Surveys, Vol.25, No.1, 1993.

[Belloir] Belloir, Bruel, Barbier. *Intégration du test dans les composants logiciels*. Workshop "OCM dans l'ingénierie des SI" during INFORSID 2002, Nantes, France, 2002.

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

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

[Favarim] Favarim, Fraga, Siqueira. *Fault-tolerant CORBA Components.* In 2nd Workshop on Reflective and Adaptive Middleware, pages 144-148, Rio de Janeiro, Brazil, 2003.

[Fraga] Fraga, Siqueira, Favarim. *An Adaptive Fault-Tolerant Component Model*. 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, Capri Island, Italy, 2003.

[Bianchini] Bianchini, Buskens. *An adaptative distributed system level diagnosis algorithm and its implementation*. Proceedings of the 21st international IEEE Symposium on Fault-Tolerant Computing, p. 616-626, 1991.

[Groß] Groß. *Component+ Methodology. Built-In Contract Testing: Technological Foundations*. IESE Report 073.02/E, Kaiserslautern, December 2002.

[Icmg] http://www.icmgworld.com.

[Kopetz] Kopetz, Tu Wien. *DECOS - European Integrated Project Proposal*. https://www.decos.at/download/021003-DECOS.Grenoble-US.pdf/, October 2002.

[Lee] Lee, Shin. *Probabilistic Diagnosis of Multiprocessor Systems.* ACM Computing Surveys, Vol.26, No.1, 1994.

[Martins] Martins, Toyota, Yanagawa. *Constructing Self-Testable Software Components*. Proceedings of the 2001 International Conference on Dependable Systems and Networks, p. 151-160, Göteborg, Sweden, July 2001.

[Marangozova] Marangozova, Hagimont. *An Infrastructure for CORBA Component Replication*. 1st IFIP/ACM Working Conference on Component Deployment, Berlin, Germany, June 2002.

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

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

[Openccm] *OpenCCM*. http://www.objectweb.org

[Prerapata] Prerapata, Metz, Chien. *On the connection assignment problem of diagnosticable system*. IEEE Transactions on Electronic Computers, vol. EC-16, n°6, p. 848-854, Décembre 1967.

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

[Wang] Wang. *On Built-In Test Reuse in Object-Oriented Frame-work Design*. ACM Computing Surveys, 32(1), March, 2000.