Software Component Diagnosis Service: Architecture Description

Thi Quynh BUI
LCIS Laboratory
INP Grenoble
Valence, France
Thi-Quynh.Bui@esisar.inpg.fr

Oum-El-Kheir AKTOUF
LCIS Laboratory
INP Grenoble
Valence, France
Oum-El-Kheir.Aktouf@esisar.inpg.fr

Michel DANG
LCIS Laboratory
INP Grenoble
Valence, France
Michel.Dang@inpg.fr

Abstract — **Component-based software development paradigm enables the construction of complex applications by assembling existing self-contained components. The cost of the software development process can then be sharply reduced. 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 paper, we propose a diagnosis service architecture that enhances dependability of component-based applications, especially embedded ones. This diagnosis service provides the ability of detection and location of faulty components during runtime system execution. As far as we know it is the first work that applies system-level diagnosis in embedded software component based applications for increasing the reliability of the whole system.**

Keywords-Dependability; diagnosis; middlewares; embedded systems; fault tolerance.

I. INTRODUCTION

Software component technology is used increasingly to develop complex software systems, for enhancing composition, reuse, modularity and configurability. Software systems are becoming more distributed and operating in highly dynamic environments. Therefore, dependability of component-based applications is an important research issue.

In this context, a fault tolerance mechanism is necessary for enabling a correct service delivery of the system. Our proposed approach is based on the diagnostic technique that concerns the ability of fault-free components to determine the fault-state of the whole application. Advantages of our architecture are application autonomy, cost-effectiveness and better usage of system resources. Such advantages are very important for embedded systems.

System-level diagnosis is a process initially introduced by Preparata *et al.* [1] 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 are called self-diagnosis algorithms. Depending test are assignment, 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.

The proposed diagnosis service provides the ability of detection and location of faulty components during runtime system execution. Once the diagnosis process is completed, the system is able to isolate the diagnosed faulty components, by ignoring their output and initiating a reconfiguration operation such that the reliability of the system can be maintained in the long run.

This paper describes the architecture of our diagnosis service based on inter-component testing. Previous results, published in [2, 3], describe the implementation of inter-component tests and the basis functioning of the diagnosis service. It is expected that the diagnosis approach should enhance application dependability with a competitive cost-performance trade-off. To support this statement, we conduct three experimental case studies, in a banking, an air conditioning and an automotive systems. Our aim is to tackle examples in embedded domain to gain a proof of concept illustration. 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 the global architecture of the proposed diagnosis approach. Section 3 describes the diagnosis service functioning. The obtained results are briefly presented in Section 4. Finally, Section 5 gives some concluding remarks.

II. DIAGNOSIS SERVICE ARCHITECTURE

Our proposed approach for diagnosing faulty components consists of two main aspects. The first one concerns the execution of the on-line inter-component tests, *i.e.* testing a component to serve the diagnosis process during the application execution. 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.

The problem we are investigating consists of proposing such an approach in an embedded software component-based application. In such application, memory usage has very strict constraints. For diagnosis purposes, it 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.

In this section, we present the proposed architecture for our approach and cite the potentially raised problems when building such architecture.

A. Global diagnosis service architecture

The global architecture is described in Figure 1.

Following, is a description of its main components:

- *AppComponent_i*: are components of the application providing testing ability.
- *Graph Partition*: divides the system into diagnosis groups where inter-component tests are performed following a given test assignment. The partitioning approaches perform better than non-partitioning ones that cover the whole system by reducing unnecessary extra message traffic and processing time [4].
- *Diagnosis Algorithm*: provides different diagnosis algorithms, that may be selected by the application.
- *Resource Monitor*: observes the usage of resources of the system (CPU, memory, bandwidth, etc) in order to provide the diagnosis process with available system resources.
- *Scheduler*: it schedules the inter-component tests with regards to the available resources and the test strategy.

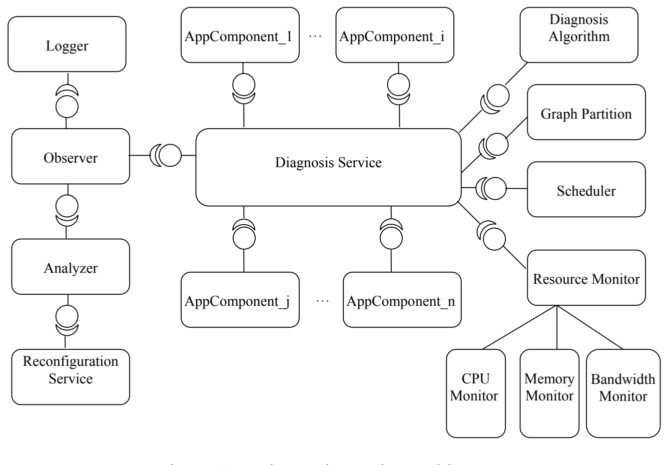

Figure 1. Diagnosis service architecture

- *Observer*: gets the global fault state of the diagnosis group or the fault state of the whole system.
- *Logger*: takes the responsibility of error logging, for statistic analysis, system report, etc. This logger is useful for the purpose of group partitioning and the choice of an appropriate diagnosis algorithm. For example, we can group the frequently faulty components in a group and use an appropriate diagnosis algorithm more efficiently, or use weightless diagnosis algorithm for the less faulty group, etc. This statistic information can also be used to determine the good diagnosis period or diagnosis time.
- *Analyzer*: receives information on the faulty component transmitted by the observer. This information, such as error logging, test cases, etc, will be analyzed for determining the type of the fault produced by the component and triggering the reconfiguration process.
- *Reconfiguration Service*: provides different strategies of system reconfiguration, for example, online repair, component replacement or system shutdown, etc.

More details on our architecture are presented in [2].

B. Potentially raised problems

As our architecture is based on online inter-component testing, in practice, we need to study the following problems due to runtime system execution:

- (1) Test implementation: how to test a component and what to test? A component is considered as a black-box with provided interfaces and required interfaces. We integrate a test interface into components to provide testing facilities.
- (2) Concurrency: tests can interfere with the business functionality of the component. What should the component do if during the testing process it receives a regular business service request from other components?
- (3) Correctness of system states: on-line testing is carried out in the real environment. If the test manipulates and changes functional data of the component, it has to provide some roll-back mechanisms, ensuring the affected data during the test are set to the original state to keep the correctness of the system state.
- (4) Efficiency: Diagnosis approach has to be efficient, meaning the approach should well use the available resources to reduce the impact on system performance. In particular, the inter-component tests need to regard the system resource, i.e. memory, CPU time or bandwidth.

In the next section, we describe the procedure of our diagnosis service and solutions to tackle the problems listed above.

III. DIAGNOSIS SERVICE PROCEDURE

The procedure of diagnosis service in the system consists of the following steps: the implementation of test interface and test cases (Subsection A), starting the diagnosis (Subsection B), diagnosis service execution (Subsection C) and end of diagnosis service (Subsection D). We will point out, during these steps, the raised problems cited in the previous section and discuss solutions for them.

We use the banking system to illustrate our concepts. This system 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.

A. *Test interface and test cases*

The first problem of section II.B (test implementation) is studied in our previous paper [3]. The principle of our component testing approach is to integrate a test interface in the component. This interface provides testing facilities and the test cases of the component.

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. So the tester component can use this interface in the same way as the other functional interfaces (Figure 2).

Generally, a component can be viewed as a state machine and requires state-transition testing. The state machine of the code validation component is illustrated in Figure 3.

The card will be locked after three unsuccessful attempts of providing a pin. 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. The other states "*SecondAttempt*" and "*ThirdAttempt*" show the second and the third attempts of the user, respectively.

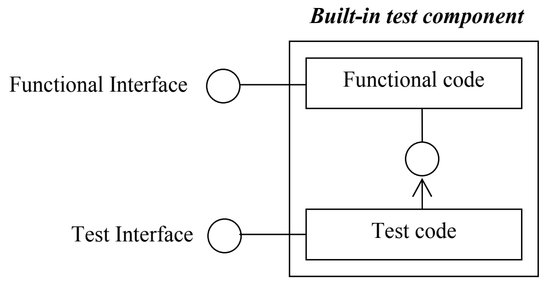

Figure 2. Built-in test component

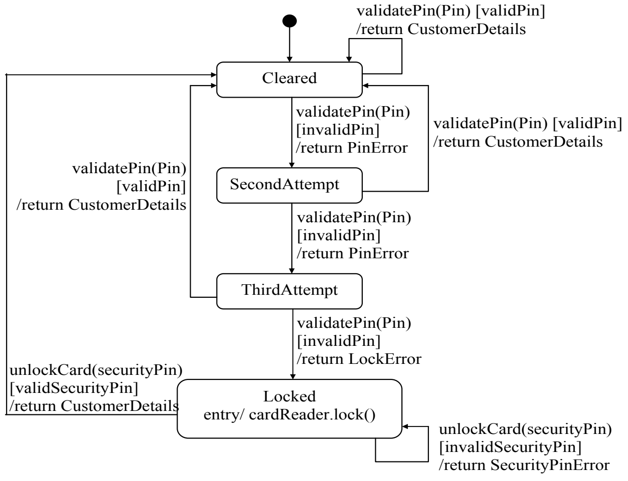
validatePin(Pin) [validPin] /return CustomerDetails
Cleared
validatePin(Pin) [invalidPin] /return PinError
validatePin(Pin) [validPin] /return CustomerDetails
validatePin(Pin) [validPin] /return CustomerDetails
SecondAttempt
validatePin(Pin) [invalidPin] /return PinError
ThirdAttempt
validatePin(Pin) [invalidPin] /return LockError
unlockCard(securityPin) [validSecurityPin] /return CustomerDetails
Locked
entry/ cardReader.lock()
unlockCard(securityPin) [invalidSecurityPin] /return SecurityPinError

Figure 3. State machine of code validation component

For component 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 [5], as shown in 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(state)* and *isInState(state).*

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

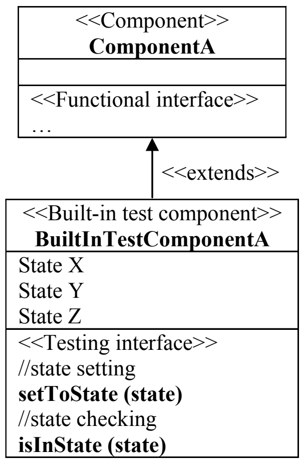

Figure 4. Concepts of built-in test component and testing interface

TABLE I. TEST CASE OF CODE VALIDATION COMPONENT

Test case	Input	Initial state	Tested method	Final state	Outcome
1	Valid Pin	Cleared	validatePin(Pin)	Cleared	CustomerDetails
2	Invalid Pin	Cleared	validatePin(Pin)	SecondAttempt	PinError
3	Valid Pin	SecondAttempt	validatePin(Pin)	Cleared	CustomerDetails
4	Invalid Pin	SecondAttempt	validatePin(Pin)	ThirdAttempt	PinError
5	Valid Pin	ThirdAttempt	validatePin(Pin)	Cleared	CustomerDetails
6	Invalid Pin	ThirdAttempt	validatePin(Pin)	Locked	PinError
7	Valid security Pin	Locked	unlockCard (securityPin)	Cleared	CustomerDetails
8	Invalid security Pin	Locked	unlockCard (securityPin)	Locked	SecurityPinError

The test cases of code validation component, corresponding to all possible transactions of the component state machine, are described in Table 1.

More details on the testing interface and how to make test cases are described in [3].

The problem of concurrency (the second problem in Section II.B) between testing and normal behavior of a component can be raised here. Concurrency occurs when the test and another component call the function simultaneously. To assure the correctness of the systems (the third problem in Section II.B), the business functionality should not be disturbed. In the literature, there exist some solutions for both problems [6]:

- The component is blocked during the testing process. While the test is executed, the test and business requests received from other components are delayed until the end of the testing process. The system correctness problem can be resolved here by restoring the component to its original state after the testing process is finished.
- The component aborts the testing process. Every time another component requests the business service of this tested component, the testing process is aborted. Once aborted, the state of the component has to be restored, the testing has to be re-launched, and the tester component that tests this component has to wait until the requested test completes.
- The component is cloned by the infrastructure before the test starts. The testing process is performed with the clone and the service process is performed with the original component. In this solution, the system correctness problem can be ignored. Besides, this option can be very resource consuming.
- The test interface of component provides methods that allow test sessions. These methods ensure that test data and real data are not mixed during the testing process. This could also be done through cloning (driven by the component itself) or by providing specific operations. In this solution, the problem of the correctness of the system state is resolved. This option puts an additional burden on the component developer.

It is worth noting that there is no optimal solution for figuring out this problem. Moreover, in embedded systems, resources are limited and our focused embedded systems are not real time systems. As a result, the first solution is probably the most interesting one. Therefore, we decided to use this solution in our diagnosis service to overcome concurrency problem.

Our implementation of this solution is straightforward by using the concurrency mechanism of the underlying programming language (i.e. threads in Java). In the test cases, when the testing process 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 the system state.

B. *Starting the diagnosis*

Our diagnosis service is started in one of these conditions:

- Change of system topology: if the topology changes (components are replaced, removed or added), the diagnosis service is triggered to verify whether the system works correctly.
- Periodicity: the diagnosis service is performed periodically according to a predefined instant.
- Admin demand: the administrator can execute the diagnosis to have the whole system state at anytime.
- Resource constraint: when the system has sufficient available resources, the diagnosis service can be launched.

These scenarios can be chosen by the user during the system configuration. One or more scenarios can be combined at the same time.

C. *Diagnosis service execution*

After triggering the diagnosis service, it will work as shown on Figure 5.

First of all, the diagnosis service uses the graph partition component to divide the application into groups in order to reduce unnecessary extra messages traffic and processing time in large systems (1). The diagnosis service loads the diagnosis algorithm chosen by the user for each group or the whole system (2). The resource monitor component observes the resources and sends to the diagnosis service information on

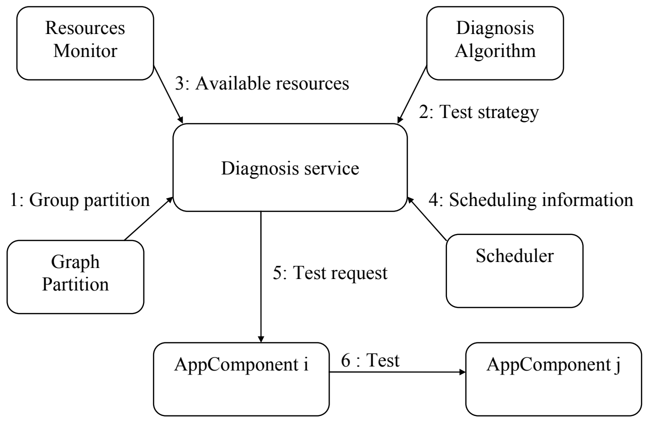

Figure 5. Diagnosis process

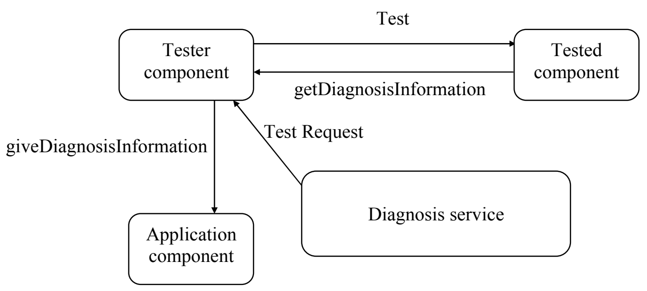

Figure 6. Interactions between diagnosis and member group components

available resources (3). Then the diagnosis service uses the scheduler component to schedule the inter-component tests (4) regarding the test strategy and the available resources of the system (and also the last test result if the diagnosis algorithm is adaptive). The test request is sent to the application component (5). This tester component analyzes this test request to know which component it will test with which test case, etc (6).

The test request can be implemented in the form of an array as follows:

Tester component	Tested component	Test cases	Send result to components

For example, the banking system uses the centralized diagnosis algorithm [4], that is based on a central reliable component (the consultation component in this case) to determine the global state of the system.

Deposit	Code validation	1,2,4	Consultation

This test request shows that Deposit component will test Code validation component with test cases 1, 2, 4 and send the test results to Consultation component, defined by the diagnosis algorithm.

Interactions between the member components in one group and the diagnosis service component are detailed in Figure 6. After receiving the testing request, 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.

The component which is responsible for the diagnosis process (analyzing the results of the inter-component tests) maintains a result table (called *syndrome*). After analyzing this syndrome, this component will have the fault state of its group.

To improve the efficiency of the system (the fourth problem in Section II.B), the scheduler component properly balances test execution and processing of the "normal" functionality. It employs the test execution strategies to provide an appropriate trade-off between testing and core functionality in an intelligent manner. These strategies are listed below:

- Idle: if the tested component is idle during a predefined time period, the tests are executed. If not, it has to force the tests.
- Resource threshold: the tests are executed if the resource availability exceeds a certain threshold for every individual resource during a predefined time period. In the opposite case, it has to force the tests.
- Priority: To reach the user's primary goal, the normal functionality should be impacted in a minimal way. This can be achieved by limiting the available resources for new tests with a threshold. This strategy is unable to differentiate between the load generated by the core functionality and the load due to other tests currently running. It is also unable to differentiate among functionalities of various importance. In order to be able to handle such situations, it could be useful to introduce priorities for the different functional requests as well as for the test requests. If priorities are set explicitly for both normal functionality and test requests then priority-based test scheduling is possible.

These strategies are chosen depending on the way the diagnosis service starts (Subsection B). If the resource constraint is decided by the user to start the diagnosis service, the scheduler will use the resource threshold for scheduling the test. In the other cases, the first strategy is preferred.

D. End of the diagnosis service

The end of the diagnosis service is described in Figure 7. The test assignment depends on the diagnosis algorithm chosen by the user. The components which make the diagnosis (by analyzing the results of the inter-component tests) are specified, and the diagnosis service informs the observer which components are responsible of the diagnosis process (1). After that, the observer connects to these components to get the fault state of the group and of the system (2). It informs the analyzer and logger components about the faulty components, for example the component Code validation is faulty with test case 4 (3). The analyzer will analyze this information and trigger the reconfiguration service (4) that can provide these reconfiguration strategies:

- In certain non-critical systems even a faulty component might be acceptable and the component will be used anyway. A warning to the user or to the log can be potentially issued to provide valuable data for debugging and problem analysis.
- The faulty component can be repaired online.

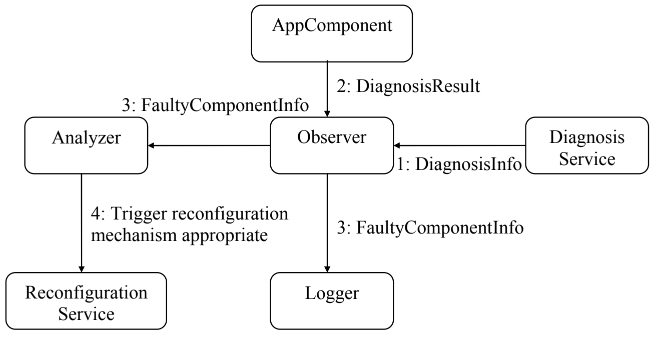

Figure 7. End of the diagnosis service

- The faulty component is replaced by another version or by another component providing the same service.
- Shut-down the system if the fault is not acceptable or in case of critical systems.

In the literature, there exists a lot of works on system reconfiguration [7]. However, most of them are based on component replacement. The principle is to replace the faulty component by a new version or a new component that provides the same service. And then, the state of faulty component is transmitted to the new component, followed by corresponding rules. The implementation of this state transfer mechanism also uses two operations *isInState(state)* and *setToState(state).* The first operation verifies whether the faulty component is residing in a logical state. The second one sets the new component to the state of the component to be replaced.

In our approach, these two operations are added in the testing interface. As a result, there is no impediment in implementing the system reconfiguration.

IV. Experimentation

We implemented the proposed approach in different embedded applications, while using different component technologies. Beside the banking system presented in the previous section, we carried out two others case studies: an air conditioning system and an automotive system.

The air conditioning system example 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 automotive system example simulates a vehicle running on a partly frozen road and comprises 6 components:

- Four independent electrical motors are put on each of the wheels. These motors can speed up the vehicle, slow it down or throw out of gear.
- One component simulates the frozen road or macadam.
- Chassis component: corresponding to the chassis of the car. It manages the total behavior of the vehicle according to the road state and wheels state.

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 [8]. The second one is a centralized algorithm that relies on a central component to determine the fault state of the whole system [1].

In practice, there are several models for component-based software development, such as Enterprise JavaBeans [9], Microsoft .Net [10], OSGi [11] and the CORBA Component Model [12]. They are being widely used and have shown improvements in the software development and maintenance process. In our work, the barking and air conditioning systems are implemented with the OpenCCM platform [13] using the CORBA Component Model, while the automotive system is realized with OSGi technology (more concretely, on the Oscar platform [14]). The centralized diagnosis method and the distributed diagnosis method are executed for a comparison purpose.

The used architecture of the diagnosis service is not as complete as in Figure 1. We implemented only the most significant components. The resource monitor and scheduler are not present in the current implementation, but are being integrated.

The obtained results measured on the banking system and air conditioning system are shown in Table II and Table III.

TABLE II. 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

TABLE III. 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

In these examples, the number of needed components consists of four system components, plus one service component 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 [15] which states that for a system with n components:

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

Table IV presents the results of the automotive system example:

TABLE IV. THE OBTAINED RESULTS OF THE AUTOMOTIVE SYSTEM

Criterion	Distributed diagnosis method	Centralized diagnosis method
Number of components needed	7	8
Fault number (t)	$5 \geq t$	$2 \geq t$
Fault detection time (ms)	88.16	39.94

Similar to the two first case studies, we have, in total, 7 components in the distributed diagnosis method and 8 components in the centralized one. The maximum fault number is calculated with the same formula.

The results in Table II, Table III and Table IV show that, there should be a trade-off in resource utilisation between diagnosis methods. The distributed method can tolerate more faulty components, yet needs more time to detect faults.

Besides integrating the lacked components in the architecture, our on-going work takes into account 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.

V. Conclusion

In this paper, we presented the main problems around the development of the proposed diagnosis service architecture, and solutions to solve these problems. The proposed inter-component and diagnosis approaches are very interesting functionalities since they may enhance application dependability with a competitive cost-performance trade-off. We are currently implementing usage of idle time and available resources to perform the tests. In parallel, we are studying efficient partitioning algorithms in order to reduce the fault detection time and improve the impact of the diagnosis method on system performances. Results of these studies will establish a good proof of our proposed approach.

A natural extension of our work is to provide a complete fault tolerance method from the diagnosis point of view. The diagnosis process will be followed by a system reconfiguration, so the reliability of the system can be maintained in the long run.

References

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

[2] T.Q. Bui and O. Aktouf, “Diagnosis service for embedded software component based systems”, Proceeding of the Second International Workshop on Engineering Fault Tolerant Systems, pp. 14-19, Dubrovnik, Croatia, September 2007.

[3] T.Q. Bui and O. Aktouf, “On-line testing of software components for diagnosis of embedded systems”, Proceeding of the 4th International Conference on Embedded and Real-Time Computing Systems, pp.330-336, Volume 22, Prague, Czech Republic, July 2007.

[4] M. Barborak and M. Malek, “Partitioning for efficient consensus”, IEEE 26th Hawaii International Conference on System Technology Sciences, pp. 438-446, June 1993.

[5] H. G. Groß, “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.

[6] D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes and R. Malaka, "The MORABIT approach to runtime component testing", pp. 171-176, 30th Annual International Computer Software and Applications Conference (COMPSAC'06), Chicago, September, 2006.

[7] A. Ketfi, N. Belkhatir, “A metamodel-based approach for the dynamic reconfiguration of component-based software”, The Eighth International Conference on Software Reuse, Madrid, Spain, July 2004.

[8] R. Bianchini and R. W. Buskens, “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.

[9] Sun Microsystems, “Enterprise JavaBeans specification”, v2.0, 2001, available: http://java.sun.com/ejb/.

[10] Microsoft, “Overview of the .NET framework”, MSDN Library White Paper, 2001, available: http://msdn.microsoft.com.

[11] OSGI, available: http://osgi.org.

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

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

[14] Oscar, available: http://www.oscar.objectweb.org.

[15] M. Barborak, M. Makek and A. Dahbura, “The consensus problem in fault-tolerant computing”, ACM Computing Surveys, Vol.25, No.2, pp. 171-220, June 1993.