A Distributed Test Architecture for Adaptable and Distributed Real-Time Systems

Mariam Lahami, Moez Krichen and Mohamed Jmaiel

Research Unit of Development and Control of Distributed Applications
National School of Engineering of Sfax, University of Sfax
Sokra road km 4, PB 1173 Sfax, Tunisia
mariam.lahami@redcad.org,moez.krichen@redcad.org
mohamed.jmaiel@enis.rnu.tn,
http://www.redcad.org

Abstract. This work focuses on testing the consistency of distributed real-time systems when their configurations evolve dynamically, called also adaptable systems. In this context, runtime testing which is carried out on the final execution environment is emerging as a new solution for the validation of these systems. To reduce testing effort, cost and time, we apply the dependency analysis technique in order to identify affected parts of the system under test due to runtime reconfiguration. In addition, we propose a flexible and evolvable distributed test architecture made of two kinds of testers: *Single Component Testers* and *Component Composition Testers*. These testers execute unit tests (respectively integration tests) on the affected components (respectively component compositions) as soon as reconfiguration actions occur. An illustrative example describing interactions between the proposed testers when two reconfiguration scenarios happen is given.

1 Introduction

Distributed real-time systems become increasingly important in a wide range of application fields, e.g. in the system automation, aerospace, robotics and so on. A real-time system is defined in Schutz (1993) as a system which has to adhere not only to functional requirements but also to temporal requirements, often called "timing constraints" or "deadlines". If a computation is activated by a stimulus from the environment, it must be completed before the specified deadline. Therefore, the system correctness depends not only on the logical results of a computation but also on the time at which the results are produced.

Currently, real-time systems are being implemented as distributed systems. This issue is manifested by an execution environment composed of multiple nodes distributed among the network. This need is due to the increasing complexity of modern applications that require more computational resources.

To guarantee their high availability at runtime, these systems are becoming highly adaptive and reactive. They need to change their configuration dynamically in order to achieve new

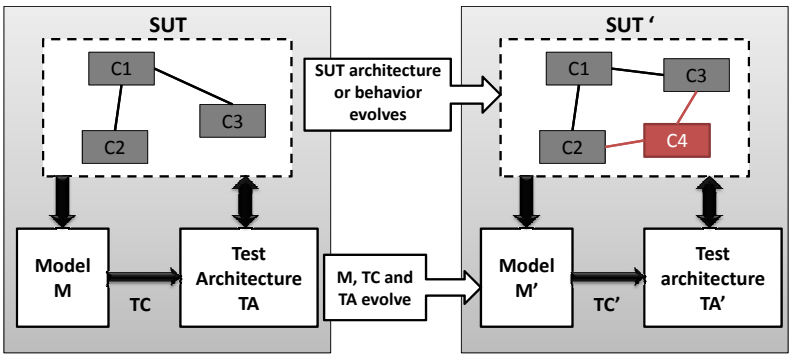

FIG. 1 – *Main objectives of our study.*

requirements and avoid failures without service interrupting. Therefore, they evolve continuously by integrating new components, deleting faulty or unneeded ones and substituting old components by new versions at runtime. Several approaches have been proposed for developing dynamically distributed real-time systems such as Schneider et al. (2004); Rasche and Poize (2005). They focus essentially on how to adapt these systems at runtime without paying attention to the system safety and consistency during and after runtime reconfiguration.

To cope with this challenging issue, a validation technique, such as testing, has to be applied in order to check functional and non-functional requirements after each dynamic reconfiguration. Nevertheless, traditional testing techniques cannot be done for these highly evolvable systems since they are applied during the development phase. For this reason, runtime testing is emerging as a new solution for the validation of the above systems. Runtime testing is defined in Brenner et al. (2007) as any testing method that has to be carried out in the final execution environment of a system while it is performing its normal work.

Various research efforts have addressed the runtime testing of evolvable systems such as *ubiquitous systems* Merdes et al. (2006), *publish/subscribe systems* Piel et al. (2010), *CBA* [1] Piel and González-Sanchez (2009); Gonzalez et al. (2008a); Niebuhr and Rausch (2009) or *SOA* [2] applications Bai et al. (2007) and *autonomic systems* King et al. (2011). Each of them proposes a runtime testing method adequate to the adopted domain that checks especially functional aspects. Non functional properties such as temporal constraints are ignored. In addition, they did not concentrate on proposing an explicit test architecture that evolves when the system under test evolves too. Also, they did not propose ways to minimize the number of testers to be deployed and the number of test cases to be re-executed after runtime reconfiguration.

We have found a lot of contributions in the area of distributed test architectures for testing distributed real-time systems. We discuss some selected research in this matter such as Tarhini and Fouchal (2005); Khoumsi (2001a); Siddiquee and En-Nouaary (2006). They propose either centralized or distributed test architecture to master testing challenges of this kind of systems. Despite the advantages of these methods, we notice that they are not suitable

1. Component Based Architecture
2. Service Oriented Architecture

for adaptable systems as they are applied only at development time. Also, the proposed test architectures are static and do not evolve when system configuration changes.

Consequently, we notice that runtime testing of Adaptable Distributed Real-Time (ADRT) systems is still one of the open issues in software engineering. The biggest testing challenge for these systems is how to check whether the system still behaves as intended during and after dynamic reconfiguration and how to guarantee that undesirable behaviors or delays have not been introduced due to runtime testing. Therefore, we propose a novel approach that is illustrated in Figure 1 with the purpose of resolving these challenges. Our work consists in associating to the SUT, a model (M) to describe its behavior, a set of test cases (TC) to execute using a specific test architecture (TA) in order to validate SUT consistency. When the SUT evolves, the behavioral model, the test cases and the test architecture have to evolve too. The main questions to be tackled in this study are the following:

- How to build a test system that facilitates the test management of dynamically evolvable systems? Since the classical test architectures, presented in Tarhini and Fouchal (2005); Khoumsi (2001a); Siddiquee and En-Nouaary (2006), have not been suitable to our context, we have to propose a test system that follows dynamic reconfiguration scenarios by adding or removing testers dynamically. In addition, we have to minimize the number of testers to be deployed and the number of test cases to be re-executed with the aim of reducing testing costs and system overheads.
- Which mechanisms are required to propagate automatically SUT changes to its behavioral model? Behavioral reconfigurations such as changing old versions by new ones can affect the behavior of the system. Therefore, we may propagate these changes to the system behavioral model automatically in order to preserve the conformance between the running system and its formal specification.
- How to obtain the adequate test cases to execute when the system evolves? We identify two challenging issues in this context. We have to generate new test cases from the evolved behavioral model if new functionalities are added to the running system. Also, we might select test cases from the existing ones or update them in the case of structural changes occurrence.
- How to apply efficiently runtime testing without affecting the behavior of the running system or its performance? We have to use a *test isolation mechanism* in order to ensure that the testing processes, which are executed in parallel with the business processes, will not disturb the running system.

In this study, we mainly address the first concern by defining an evolvable distributed test architecture with the purpose of runtime testing management. The remaining of this paper is structured as follows. Section 2 presents some basic concepts such as the architecture of distributed real-time systems, the definition of dynamic reconfiguration and its different kinds and also the definition of runtime testing techniques. Finding dependencies in the system under test and computing its affected parts by the runtime reconfiguration is illustrated in section 3. Section 4 presents the distributed runtime testing architecture and its constituents. In section 5, an illustrative example is given to show the cooperation and the communication between the proposed testers. A brief description of related work is addressed in the section 6. Finally, section 7 concludes the paper and draws some future work.

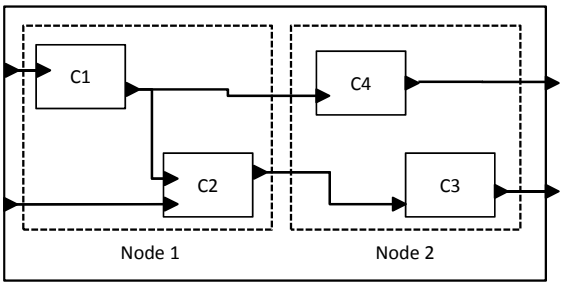

FIG. 2 – *Distributed Real-Time system architecture.*

2 Background

In this section, we investigate some basic concepts. First of all, we give an overview of the main characteristics of a distributed real-time system. Subsequent, we define the notion of dynamic reconfiguration and we introduce its different kinds. Next, we present the testing technique applied for validating functional and non-functional requirements of highly dynamic systems, called runtime testing. Finally, we introduce some techniques used to isolate runtime testing processes from the running SUT.

2.1 Overview of Distributed Real-Time systems

In this section, we focus on describing the software architecture of Distributed Real-Time (DRT) systems at a high level of abstraction. As explained in Wall (2000); Rastofer and Bellosa (2001), the real-time system architecture consists of software components and their interconnections. Each software component contains a number of input and output ports. It can be either simple or composed of subcomponents. It communicates with other components by sending events or messages through its output ports and receives from them events or messages on its input ports. We should mention that there is no restriction on how many input ports may be connected to an output port and vice-versa.

To handle reuse and distribution issues, we suppose that components can be reused in different parts of the system and can be distributed among nodes. We follow here UML notation by using the concept of node to represent physical devices such as processors. In Figure 2, the structure of a system consisting of four components is displayed. The arrows between the components represent function calls through component input and output ports.

Behavioral Specifications of such systems can be obtained using different formalisms either formal (e.g. timed automaton, Petri net) or semiformal (e.g. UML). We have to mention that modeling DRT systems behaviors is out the scope of this paper. We have only concentrated on formalizing system architecture and its components dependencies.

In our context, the adopted architecture and its behavioral model may evolve during runtime in order to achieve new requirements and to provide the greatest flexibility to the distributed real-time system. This runtime evolution which is also called dynamic reconfiguration has to take place without stopping the system. This issue is discussed with further details in the subsection below.

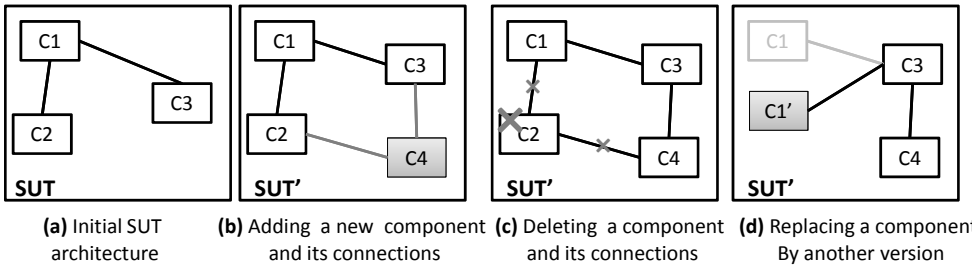

FIG. 3 – *The basic reconfiguration actions.*

2.2 Possible reconfiguration scenarios

Dynamic software reconfiguration is a useful tool to adapt and maintain software systems. It is defined in Kramer and Magee (1985) as the ability to modify and extend a system while it is running. Several modifications could be made, such as adding, removing or replacing components, or changing connectors, without stopping the system in order to meet the evolving human needs and the technology as well as the environment changes.

In the context of DRT systems, dynamic reconfiguration can be a mean to achieve flexible and dependable systems. In fact, adapting them allows avoiding system degradation, ensuring temporal correctness and achieving fault tolerance requirements. Depending on the motivation leading to the change, we can generally distinguish four categories of changes as defined by Ketfi et al. (2002):

Corrective reconfiguration : removes the faulty behavior of a running component by replacing it with a new version that provides exactly the same functionality. For instance, if a component missed its specified deadlines, it must be replaced with a correct one able to continue the task currently done by the faulty component.

Extending reconfiguration : extends the system by adding new components or new functionalities in their implementation to satisfy new emerging requirements.

Perfective reconfiguration : aims to improve the system performance even if it runs correctly. For example, we may replace a component with a new one that has more optimized implementation.

Adaptive reconfiguration : allows adapting the working system to a new running environment.

Some possible reconfiguration actions that are adopted in this work are depicted in Figure 3. These actions may be classified into two categories:

Actions on components:

- *Add_Component:* affects the structure of the system by adding a new component in a target node at runtime.
- *Delete_Component:* affects the structure of the system by removing components from the system at runtime.
- *Move_Component:* it allows component migration from an execution node to another. Hence, communications between the moved component and other components should be adapted according to the new location.

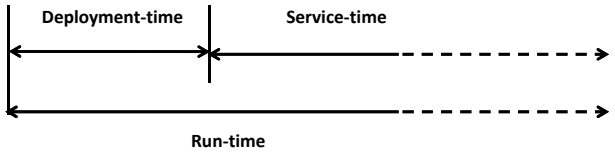

FIG. 4 – *The run-time level in software life cycle.*

– *Replace_Component:* replaces a component version by another in order to enhance the system performance.

Actions on connections:

– *Add_Connection:* affects the structure of the system by establishing local or remote connections between two components at runtime.
– *Delete_Connection:* affects the structure of the system by removing local or remote connections from two components at runtime.

By reconfiguring dynamically DRT systems, problems may arise such as incompatibilities between components, undesirable behaviors, missing deadlines, performance degradation, etc. In these situations, it is crucial to investigate ways to dynamically validate these systems with the aim of avoiding the high costs of system failures. Therefore, we adopt runtime testing as a novel validation technique in order to detect as soon as possible the above mentioned problems.

2.3 Runtime Testing Of ADRT Systems

Currently, runtime testing is emerging as a novel solution for the validation of adaptable systems. It is defined in Brenner et al. (2007) as any testing method that is carried out on the final execution environment of a system. As shown in Figure 4, it can be performed first at deployment-time and second at service-time. The deployment-time testing serves to validate and verify the assembled system in its runtime environment while it is deployed for the first time. For systems whose architecture remains constant after initial installation, there is obviously no need to retest the system when it has been placed in-service. On the contrary, if the execution environment or the system behavior otherwise its architecture has changed, service-time testing will be a necessity to verify and validate the new system in the new situation.

As previously mentioned in the definition of runtime testing, any test method can be applied at runtime such unit testing, integration testing, regression testing, etc.

Unit testing is used to validate that the component behavior still conforms to its specification while it is running in isolation in the execution environment.

Integration testing is used at runtime to validate that the affected component compositions by the reconfiguration action still behave as intended.

Regression testing as defined in Harrold (2000) “attempts to validate modified software and ensure that no new errors are introduced into previously tested code ”. This technique guarantees that the modified program is still working according to its specification and it is maintaining its level of reliability. When the program code is modified (e.g. behavioral changes), the regression testing is called *code-based regression testing* (code-based RT). We have also identified another kind of regression testing called *software architecture based regression test-*

ing (SA-based RT) as illustrated in Muccini et al. (2006). It consists on reiterating the testing process in a cost effective manner whenever the software architecture of a system is modified over time (e.g. architectural changes).

In our work, we aim to apply unit and integration tests only on the affected parts of the ADRT system in order to minimize the number of testers to be deployed and the number of test cases to be re-executed. Consequently, the testing effort, cost and time will be reduced. However, other challenges still persist such as test processes interference with the business processes of the running system due to their parallel execution. The next subsection introduces some mechanisms resolving this problem.

2.4 Test Isolation Mechanisms

While testing at runtime the SUT, business and test processes are running in parallel. Consequently, Test cases could interfere with its business functionalities and affect its other clients or sub-systems. The best way to resolve the interference problem between test processes and business processes of the SUT is the application of test isolation mechanisms widely discussed in Piel et al. (2010); Gonzalez et al. (2008b) and listed below :

- *Duplicating the SUT* is a technique which consists in replicating the SUT for testing the new configuration while the previous one is running Suliman et al. (2006); King et al. (2011). Such technique is very resource consuming and can be used in virtual environments such as cloud or grid where computational resources are available for the test system.
- *Blocking the SUT* during the hole testing process. As a result, business requests received from other components are either rejected or delayed until the end of the testing process Suliman et al. (2006); King et al. (2011).
- *Tagging test data* technique consists in adding a special flag to the testing data in order to be able to differentiate it from the business data Piel et al. (2010).
- *Built-in Test* technique is mainly used for equipping components with special interfaces in order to facilitate the testing activity before or during runtime Suliman et al. (2006); Brenner et al. (2007).

3 Finding System Under Test Dependencies

As mentioned in the section above, the system under test (SUT) is modeled as a set of communicating components that collaborate together to fulfill some specified behaviors. We assume that components dependencies have been explicitly defined without having knowledge of the internal implementation of each component. Dependencies between components is defined in Alhazbi and Jantan (2007) as “the reliance of a component on other(s) to support a specific functionality”. It is also considered as binary relation between two components: antecedent and dependent as illustrated in Figure 5.

- Antecedent is the independent component that has an impact on the dependent one if it is deleted or altered.
- Dependent is the component that is related to its antecedents where dynamic changes in them might lead to dependent component failures.

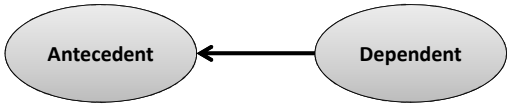

FIG. 5 – *Dependency Relationship.*

Formally, the relation $\rightarrow$ called "Depend on" is defined in Larsson and Crnkovic (2001) where $C_i \rightarrow C_j$ means that component C_i depends on the component C_j. The set of all dependencies in a component-based system is defined as:

$D = \{(C_i, C_j) : C_i, C_j \in S \wedge C_i \rightarrow C_i\}$ where S is the set of components in the system. Accordingly, the current system configuration is a set of components and its dependencies $Con = (S, D)$.

In the rest of this section, we present the model used to capture direct and indirect dependencies, on the one hand. On the other hand, we illustrate how to apply dependencies analysis when different reconfiguration scenarios occur.

3.1 Dependency representation

In order to analyze dependencies efficiently, we have to use a good formalism representing dependencies between components. Usually, dependency graph and adjacency matrix are used in many research activities such as Alhazbi and Jantan (2007); Li (2003) with the purpose of generating a minimal set of affected system parts by a reconfiguration action.

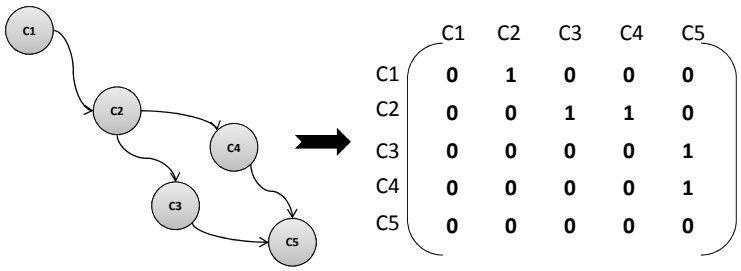

FIG. 6 – *Component Direct Dependency Graph and its Adjacency Matrix representing direct dependencies.*

In our study, we adopt the same representation model of dependencies called *Component Direct Dependency Graph* (CDDG). In fact, a CDDG is a directed graph denoted by $G = (S, D)$ where :

- S is a finite nonempty set of vertices representing system's components and
- D is a set of edges between two vertices. For instance, $(a, b) \in D$ means $a \rightarrow b$ and $D \subseteq (S \times S)$.

The Figure 6 shows an example of a CDDG and its corresponding adjacency matrix. In such matrix, called $AM_{n \times n}$, each component is represented by a column and a row. If a

component C_i depends on a component C_j then $AM\,[i,j] = 1$. Initially, AM represents only direct dependencies. In order to derive all indirect dependencies in the SUT, we have to calculate the transitive closure of the graph.

Several transitive closure algorithms have been widely studied in the literature such as Roy-Warshall algorithm and its modification proposed by Warren Ioannidis and Rantakrishnan (1988). In the worst case, both algorithms compute the transitive closure on $\theta(n^3)$ times where n is the number of vertices of the graph. This complexity has been enhanced in Jaumard and Minoux (1986) by proposing an algorithm computing the transitive closure only for cycle-free graphs on better than $\theta(mn)$ times where m is the number of edges of the graph. In our context, we adopt Roy-Warshall algorithm because it is sufficient for computing transitive closure of any graph and has an acceptable complexity. Figure7 depicts the obtained adjacency matrix that represents direct and indirect component dependencies.

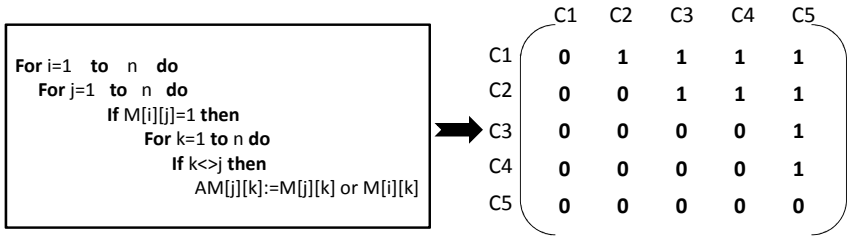

FIG. 7 – *Adjacency Matrix representing direct and indirect dependencies produced by the Roy-Warshall'algorithm.*

3.2 Dependency analysis

Analyzing components dependencies and computing the affected parts of the SUT after each dynamic change is handled in this subsection. We focus especially on three kinds of reconfiguration actions as depicted in Figure 3: adding a new component and its connections, deleting an existing component and its connections and replacing a component by another version.

3.2.1 Adding a new component and its connections

Dynamically adding a new component to the SUT architecture might affect not only its direct dependents but also indirect ones. The adjacency matrix will be modified by adding new rows and columns representing direct dependencies between the new component and the old ones. Indirect dependencies are obtained as mentioned before transitively by applying the Roy-Warshall's algorithm in order to generate an adjacency matrix defining direct and indirect dependencies. The current system configuration will be modified $New\_Con = (New\_S, New\_D)$ with:

- New_S=$S \cup C_{new}$ and
- New_D=$D \cup \{(C_{new}, C) : C_{new} \rightarrow C\} \cup \{(C, C_{new}) : C \rightarrow C_{new}\}$

Procedure 1 $AffectedComponentsIdentification\_By\_AddAction$

Input: A new component C_{new},
The new configuration $New\_Con = (New\_S, New\_D)$,
The new adjacency matrix AM.
Output: The array $AffectC$ that contains the affected components by the add action.

1: **var:**
2: Col : The column index in AM that represents the component C_{new} to add;
3: lig : The column index in AM that represents the component C_{new} to add;
4: i, j, k : integer;
5: **BEGIN**
6: **for** $i = 0$ to $New\_Con.New\_S.size - 1$ **do**
7: **if** ($New\_Con.New\_S[i]$= C_{new}) **then**
8: Col=lig=i
9: **end if**
10: **end for**
11: $k = 0$
12: **for** $j = 0$ to $New\_Con.New\_S.size - 1$ **do**
13: {Find components that depend on C_{new}}
14: **if** ($AM[j][col] = 1$) **then**
15: $AffectC[k] = New\_Con.New\_S[j]$
16: $k = k + 1$
17: **end if**
18: {Find components that C_{new} depends on}
19: **if** ($AM[lig][j] = 1$) **then**
20: $AffectC[k] = New\_Con.New\_S[j]$
21: $k = k + 1$
22: **end if**
23: **end for**
24: **END**

The Procedure 1 defines how to obtain all the affected components by the add action. It has as inputs the component to add, the new configuration and the new adjacency matrix describing direct and indirect dependencies. All the components that depend on C_{new} and C_{new} depends on are affected by this action. To calculate them, we have to search first in the adjacency matrix for the non-zero elements in the column corresponding to C_{new}. The non-zero elements indicate that the corresponding components depend on C_{new}. Second, we seek for the non-zero elements in the line corresponding to C_{new}. These elements indicate that C_{new} depends on the corresponding components.

3.2.2 Deleting an existing component and its connections

Procedure 2 $AffectedComponentsIdentification\_By\_DelAction$

Input: A component $C_{removed}$,
The old configuration $Con = (S, D)$,
The old adjacency matrix AM.
Output: The array $AffectC$ that contains the affected components by the delete action.

1: **var:**
2: Col : The column index in AM that represents the component $C_{removed}$ to delete;
3: i, j, k : integer;
4: **BEGIN**
5: **for** $i = 0$ to $Con.S.size - 1$ **do**
6: **if** ($Con.S[i]$= $C_{removed}$) **then**
7: Col=i
8: **end if**
9: **end for**
10: k=0
11: **for** $j = 0$ to $Con.S.size - 1$ **do**
12: **if** ($AM[j][col] = 1$) **then**
13: $AffectC[k] = Con.S[j]$
14: $k = k + 1$
15: **end if**
16: **end for**
17: **END**

Deleting a component from the SUT architecture at runtime might affect its direct and indirects dependents which must be tested and validated. The current system configuration will be modified $New\_Con = (New\_S, New\_D)$ with:

- New_S=$S \setminus C_{removed}$ and
- New_D=$D \setminus \{(C, C_{removed}) : C \rightarrow C_{removed}\}$

The Procedure 2 has as inputs the component to delete, the old configuration and the old adjacency matrix describing direct and indirect dependencies. As output, it computes the affected components set by this delete action. To find them, we use the adjacency matrix AM by searching the non-zero elements in the column corresponding to $C_{removed}$. These non-zero

elements indicate direct or indirect dependent components on $C_{removed}$. Consequently, those components will be affected when removing $C_{removed}$.

3.2.3 Replacing a component by another version

Replace action can be seen as a set of adding and removing components. In fact, replace an old component C_i by a new component C_i' is done by calling:

– The delete action that deletes the old component C_i and all its dependencies and

– The add action that adds the new component C_i' and all its dependencies.

Therefore, we use the two procedures defined above as illustrated in the Procedure 3. This procedure is called only when the new version has different interfaces comparing to the old one.

Procedure 3 $AffectedComponentsIdentification\_By\_ReplaceAction$

Input: The old component C_{old},
The new component C_{new},
The system configuration $Con = (S, D)$.

Output: The affected components set by the replace_Component action $AffectC$.

1: **BEGIN**
2: $AffectC$=$AffectedComponentsIdentification\_By\_DelAction(C_{old}, Con, AM) \cup AffectedComponentsIdentification\_By\_AddAction(C_{new}, Con, AM)$
3: **END**

In this section, we have concentrated on studying the dependencies between components and on proposing some procedures that facilitate the generation of affected components. We have to mention here that the generation of affected component compositions by a reconfiguration action is out of the scope of this work.

4 Distributed Runtime Testing Architecture for ADRT Systems

We propose a distributed runtime testing architecture to handle the validation of ADRT systems after dynamic reconfiguration. It consists of two kinds of testers: *Single Component Tester* and *Component Composition Tester*, coordinating together and communicating with the SUT. The overall test process is managed via a *Test System Coordinator* as shown in Figure 8.

As described in Krichen (2010), the tester is an entity that interacts with the SUT with the purpose of executing the available test cases and then observing the response of the SUT due to this excitation. In general, test cases are generated automatically from formal specifications. Usually, they are composed of two kinds of interactions:

– The outputs: modeling the observation and the sending of a message from the system.

– The inputs: modeling the sending of a message to the system.

We have to note that test cases generation from the SUT formal specification is out the scope of this current work. In the following, we describe briefly the role of each entity in the proposed test architecture:

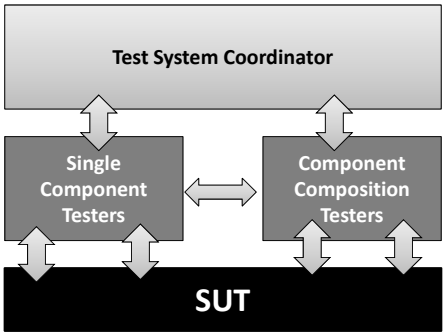

FIG. 8 – *A generic overview of distributed test architecture.*

Single Component Tester (SCT) : is dedicated to test the behavior of a single component which has been newly added to the running configuration by executing unit tests. Each SCT produces a local verdict: PASS if all the traces correspond to the test case, INCONCLUSIVE if the tester cannot execute the test case and FAIL otherwise.

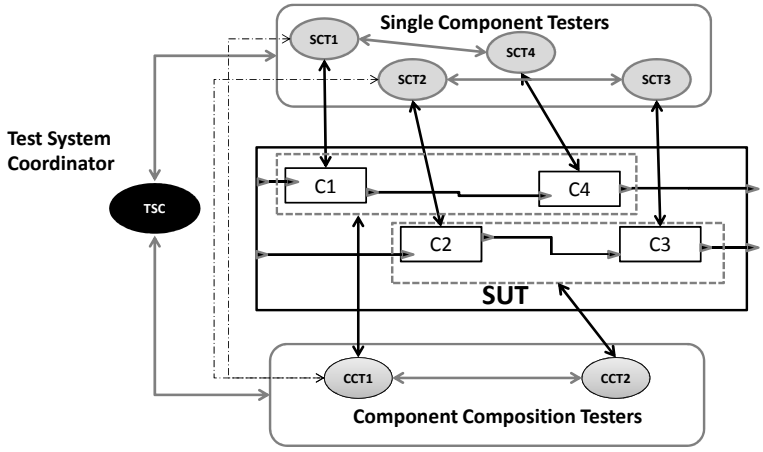

FIG. 9 – *Detailed overview of the distributed test architecture*

Composite Component Tester (CCT) : is introduced to test component composition affected by the dynamic reconfiguration. Each one executes a set of integration tests with the aim of verifying functional and timing properties. CCTs may exchange coordination messages in order to check remote connections. They may also coordinate with SCTs to produce a partial verdict. The latter depends on the local verdicts of the SCTs and the results of test cases execution that validate the affected component composition.

Test System Coordinator (TSC) : is responsible for starting the test process. It instantiates the two kinds of testers in their host computers. It can also remove them when no longer

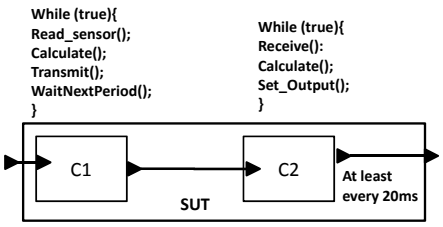

FIG. 10 – *Simple real-time application.*

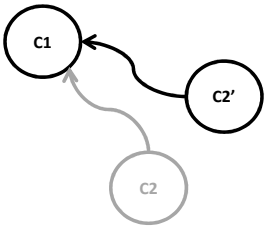

FIG. 11 – *The CDDG of the new SUT architecture after replacing a component by another.*

required. It may assign test cases for test execution and provides the global verdict. The latter is deduced from verdicts given by single component testers or component composition testers.

While the running system evolves during runtime, dependency analysis step has to be done with the purpose of computing affected components and composition by this evolution. At this time, our test architecture has to be designed in order to check the new configuration of the system. The number of single component testers as well as component composition testers to be deployed have to be fixed accordingly to the affected parts of the SUT.

We illustrate in Figure 9 the interaction between testers and SUT with further details. The solid arrows between Test System Coordinator (TSC) and testers correspond to the management actions that TSC can execute in order to master the test architecture. The solid arrows between single component testers (SCT) (respectively component composition testers (CCT)) and components under test (respectively component assemblies under test) show sending input data and receiving output results. Finally, the dashed arrows between testers illustrate their synchronization and coordination to test efficiently the SUT.

5 Illustrative example

We might mention that runtime testing can be performed by SCT or CCT testers locally (in the same host computer with SUT) or remotely (in other host computer). This depends on the available resources. By doing this, we can get confidence in ADRT systems by saving time, energy and cost required for testing. In addition, this technique can be used before or after the execution of reconfiguration actions. If test cases are executed by the related testers before the occurrence of dynamic reconfiguration, test results can be helpful to the reconfiguration manager to decide the proceeding of the reconfiguration or not. Applying runtime testing technique after reconfiguration allows detecting inconsistencies and missed deadlines. In the sequel, we depict with further details the communication between the different entities in the distributed test architecture via an illustrative example.

To illustrate tester interactions, we handle an example inspired from Rasche and Poize (2005). Figure 10 shows the initial configuration of the real-time system. One component C1 processes data of an external sensor and transmits results to a second component C2. The latter has to generate outputs signals each 20ms. In the following subsections, we study two reconfiguration scenarios: the substitution of a component by another version and the insertion of new component. For each scenario, we describe the corresponding distributed test architecture.

5.1 Substitution scenario

We suppose that the component C2 in the SUT will be replaced by another version C2' with the aim of increasing SUT performance, for instance, the generation of output signals each 15 ms. To verify that the new version C2' is conform to its formal specification (for example, timed automaton) and the new requirements will be satisfied, runtime testing is performed. To do this, we suppose that test cases are available to validate the new version C2' and an oracle is also given. Also, dependency analysis has been performed as depicted in Figure 11. In this case, the whole system is affected by this action due to the simplicity of the example.

As shown in Figure 12, interactions between testers are described in the following steps:

- The TSC instantiates a single component tester SCT1 and a component composition tester CCT1 with the purpose of testing respectively the new version C2' in isolation and the component composition affected by this reconfiguration action.
- TSC assigns the adequate test cases to the related tester. These test cases can be selected from existing ones or newly generated.
- The SCT1 can test the new version in isolation, without taking any communication input from other components. Also, testing the new version that is plugged into the system with the old version still working is possible. The local verdict of SCT1 is sent to CCT1.
- If the local verdict of SCT1 is PASS, CCT1 proceeds testing the component composition and notifies the TSC with its partial verdict. Otherwise, T1 sends the local verdict to TSC without executing its own test cases.
- TSC has to produce its global verdict with the respect to the verdicts of component composition testers.

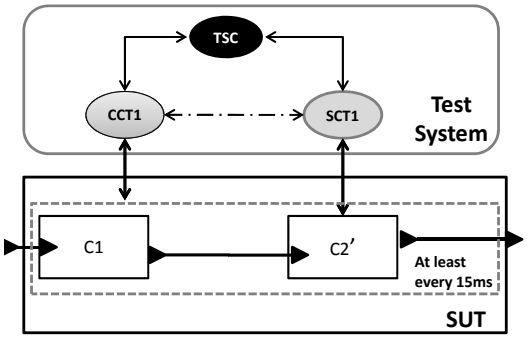

FIG. 12 – *Distributed test architecture after replacing a component by another version.*

5.2 Insertion scenario

Figure 13 shows that a new component is added to the system architecture at runtime. This component generates logging data for debugging purposes.

We have to check in this case that the new component behaves correctly to its specification. We must also verify that the overall system is not influenced by this reconfiguration and meets its deadline. For example, component C2 must generate its output signals each 20 ms. Figure

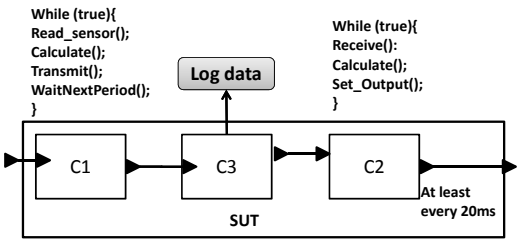

FIG. 13 – *Added debug component.*

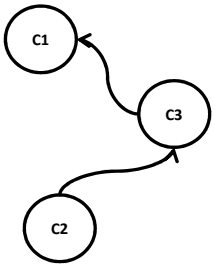

FIG. 14 – *The CDDG of the new SUT architecture after adding new component.*

14 illustrates the new component direct dependency graph after adding the new component C3 and its dependencies.

The test architecture suitable to verify the system in the case of this scenario is presented in Figure 15. We have to use a single component tester to check the behavior of the new component. As the three components in this example are affected by this runtime reconfiguration, a component composition tester has to be instantiated as well to test the overall system.

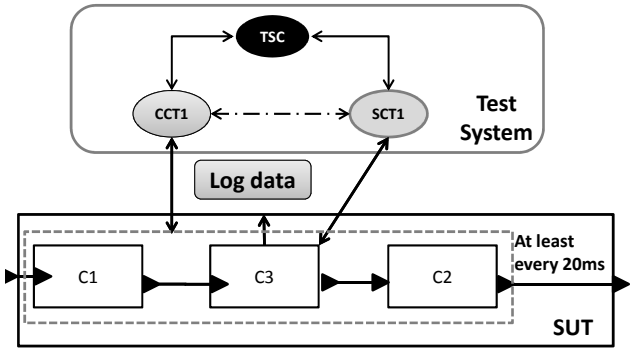

FIG. 15 – *Distributed test architecture after adding new debug component.*

6 Related Work

Recent researches have been proposed to deal with runtime testing in dynamic environment. They aim to ensure the correctness of the running system after reconfiguration. In fact, we distinguish approaches dealing with ubiquitous software systems Merdes et al. (2006), CBA systems Piel and González-Sanchez (2009); Niebuhr and Rausch (2009); Gonzalez et al. (2008a), SOA systems Bai et al. (2007), publish/subscribe systems Piel et al. (2010) and autonomic systems King et al. (2011). The majority are based on the well-known Built-In Test

(BIT) paradigm described in Wang et al. (1999). In BIT approach, tests are integrated into the component and accessible via a test interface. It is mainly designed for stable topologies and further developed by Merdes et al. (2006); Bai et al. (2007); Gonzalez et al. (2008a); Piel and González-Sanchez (2009); Piel et al. (2010) to improve the testability of dynamic applications at runtime. This paradigm facilitates the component testing by itself or by others components. However, the proposed test system will be tightly coupled with the SUT. Thus, the SUT developers will have a considerable development burden. In addition, the BIT approach is not often adequate for real-time systems especially if they are limited in computational and storage resources. Built-in test code may occupy a lot of space in the component and lead to the increase of component size and its complexity. This may affect the performance of the whole system and also violate runtime requirements in particular in terms of timing. Moreover, these approaches did not concentrate on proposing an explicit test architecture that evolves when the system under test evolves too. To the best of our knowledge, there is only one approach that incorporates dynamicity in the proposed test system Bai et al. (2007). It proposes an explicit test architecture that follows the dynamic changes on the services under test but with testing the whole system. It did not propose ways to minimize the number of testers to be deployed and the number of test cases to be re-executed after runtime reconfiguration.

Unlike these approaches, our work aims at defining distributed test architecture that conserves the current resources by instantiating testers if needed and by testing only the affected parts of the system. This has an important impact on reducing testing time and costs as well as avoiding overheads and burdens.

Proposing test architectures for managing DRT systems is carried out in several research work. They offer either centralized Khoumsi (2001b); Siddiquee and En-Nouaary (2006) or distributed Tarhini and Fouchal (2005); Khoumsi (2001a); Siddiquee and En-Nouaary (2006) test architecture for stable environments. The centralized architecture presented in Khoumsi (2001b) consists of a single tester that communicates with the different ports of the system under test. It considers that the SUT is distributed among several sites and contains a port in each of its sites. This architecture is enhanced in Khoumsi (2001a) by associating to each port a local tester and a local clock. Following the same principles, Siddiquee and En-Nouaary (2006) proposes two testing architectures. The centralized one is made of a synchronizer which embeds internal small testers and one global clock. The role of synchronizer is to execute test suites on the SUT and return a verdict about its conformance to its specification. The issue of conformance testing was considered in Tarhini and Fouchal (2005). This work proposes a distributed test architecture consisting of a set of Timed Input-Output Automata each of which represents the specification of each SUT component and a distributed tester that contains a set of coordinating testers. Each tester is dedicated to test a single SUT component.

In spite of the advantages of these methods, they may not be suitable for adaptable systems. The proposed test architectures dealt with development-time testing not runtime testing. In addition, they are not flexible and evolvable to ensure new testing requirements. If a dynamic reconfiguration action happens, it affects a part of the system not its totality. Therefore, following these approaches by associating testers to each component in the SUT will be unnecessary and will burden its performance. Testers must be added and removed to the test architecture dynamically and accordingly to the occurred reconfiguration action in order to reduce testing costs and overheads as proposed in our work.

7 Conclusion

In this article, we have dealt with runtime testing of ADRT systems. We have focused on this research area with the intention of increasing the dependability of such systems. Obtaining dependable distributed real-time systems even after dynamic reconfiguration is a very challenging issue. In this work, we aim to use runtime testing to detect system inconsistencies when reconfiguration actions happen.

To do this efficiently, we have planned to minimize the set of components to be validated after the occurrence of reconfiguration actions. For this reason, we have revealed how to analyze components dependencies in the SUT and how to compute the affected components sets by some given procedures.

In addition, we have proposed a distributed runtime testing architecture with the purpose of validating the affected subsystem. The latter contains testers that can be added or removed dynamically by following the dynamic reconfiguration applied on the system under test. The proposed architecture can be a first step for managing test cost and time with efficient manner. Communications between testers have been illustrated via a simple real-time system.

As future work, we intend to enhance the dependency analysis step by procedures that compute the affected composition by a reconfiguration action. In addition, we aim to study test isolation mechanisms that resolve some challenges due to the parallel execution of runtime testing with the SUT behavior such as interference problem between test processes and components business processes. Thus, we have to choose the best way to apply this validation technique with reducing such risk. Also, we investigate effort to distribute testers in the adequate nodes with respecting resource constraints. Moreover, we plan to implement the proposed test architecture using the standard TTCN-3 (Testing and Test Control Notation [3]). The latter will be adopted as a test notation used for writing test cases and as an execution platform used for facilitating runtime testers deployment and management. Finally, we aim to deal with test cases generation and selection while behavioral reconfigurations occur.

References

Alhazbi, S. and A. Jantan (2007). Dependencies management in dynamically updateable component-based systems. *Journal of Computer Science 3*(7), 499–505.

Bai, X., D. Xu, G. Dai, W.-T. Tsai, and Y. Chen (2007). Dynamic reconfigurable testing of service-oriented architecture. Volume 1, pp. 368 –378.

Brenner, D., C. Atkinson, R. Malaka, M. Merdes, B. Paech, and D. Suliman (2007). Reducing verification effort in component-based software engineering through built-in testing. *Information Systems Frontiers 9*(2-3), 151–162.

Gonzalez, A., E. Piel, and H.-G. Gross (2008a). Architecture support for runtime integration and verification of component-based systems of systems. In L. M. H. M. A. P. O. S. Mauro Caporuscio, Antinisca Di Marco (Ed.), *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pp. 41–48. IEEE Computer Society.

3. http://www.ttcn-3.org/

Gonzalez, A., E. Piel, H.-G. Gross, and M. Glandrup (2008b). Testing challenges of maritime safety and security systems-of-systems. In *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques*, Washington, DC, USA, pp. 35–39. IEEE Computer Society.

Harrold, M. J. (2000). Testing: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, New York, NY, USA, pp. 61–72. ACM.

Ioannidis, Y. E. and R. Rantakrishnan (1988). Efficient transitive closure algorithms. In *In Proceedings of the 14th VLDB Conference*, Los Angeles, California.

Jaumard, B. and M. Minoux (1986). An efficient algorithm for the transitive closure and a linear worst-case complexity result for a class of sparse graphs. *Information Processing Letters 22*(4), 163 – 169.

Ketfi, A., N. Belkhatir, and P. yves Cunin (2002). Dynamic updating of component-based applications. In *SERP'02, June 2002, Las Vegas*.

Khoumsi, A. (2001b). Testing distributed real time systems using a distributed test architecture. In *ISCC*, pp. 648–654.

Khoumsi, A. (May 2001a). Testing distributed real-time reactive systems using a centralized test architecture. In *North Atlantic Test Workshop(NATW)*, pp. 648–654.

King, T. M., A. A. Allen, R. Cruz, and P. J. Clarke (2011). Safe runtime validation of behavioral adaptations in autonomic software. In J. M. A. Calero, L. T. Yang, F. G. Mármol, L. J. García-Villalba, X. A. Li, and Y. Wang (Eds.), *ATC*, Volume 6906 of *Lecture Notes in Computer Science*, pp. 31–46. Springer.

Kramer, J. and J. Magee (1985). Dynamic configuration for distributed systems. *IEEE Trans. Software Eng. 11*(4), 424–436.

Krichen, M. (2010). Brief announcement: A formal framework for conformance testing of distributed real-time systems. In *Proceedings of the 14th International Conference On Principles Of Distributed Systems, OPODIS 2010*, Tozeur, Tunisia.

Larsson, M. and I. Crnkovic (2001). Configuration mannagement for component-based systems. In *The tenth International Workshop on Software configuration Management*.

Li, B. (2003). Managing dependencies in component-based systems based on matrix model. In *Proc. Of Net.Object.Days 2003*, pp. 22–25.

Merdes, M., R. Malaka, D. Suliman, B. Paech, D. Brenner, and C. Atkinson (2006). Ubiquitous rats: how resource-aware run-time tests can improve ubiquitous software systems. In *SEM '06: Proceedings of the 6th international workshop on Software engineering and middleware*, New York, NY, USA, pp. 55–62. ACM.

Muccini, H., M. S. Dias, and D. J. Richardson (2006). Software architecture-based regression testing. *Journal of Systems and Software 79*(10), 1379–1396.

Niebuhr, D. and A. Rausch (2009). Guaranteeing correctness of component bindings in dynamic adaptive systems based on runtime testing. In *SIPE 09: Proceedings of the 4th international workshop on Services integration in pervasive environments*, New York, NY, USA, pp. 7–12. ACM.

Piel, É. and A. González-Sanchez (2009). Data-flow integration testing adapted to runtime evolution in component-based systems. In H.-G. Gross, M. Lormans, and J. Tretmans (Eds.),

Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime, New York, USA, pp. 3–10. Association for Computing Machinery.

Piel, É., A. González-Sanchez, and H.-G. Groß (2010). Automating Integration Testing of Large-Scale Publish/Subscribe Systems. In A. Hinze and A. P. Buchmann (Eds.), *Principles and Applications of Distributed Event-Based Systems*, pp. 140–163. IGI Global.

Rasche, A. and A. Poize (2005). Dynamic reconfiguration of component-based real-time software. pp. 347–354.

Rastofer, U. and F. Bellosa (2001). Component-based software engineering for distributed embedded real-time systems. *Software, IEE Proceedings - 148*(3), 99 –103.

Schneider, E., F. Picioroagă, and U. Brinkschulte (2004). Dynamic reconfiguration through osa+, a real-time middleware. In *DSM '04: Proceedings of the 1st international doctoral symposium on Middleware*, New York, NY, USA, pp. 319–323. ACM.

Schutz, W. (1993). *The Testability of Distributed Real-Time Systems*. Norwell, MA, USA: Kluwer Academic Publishers.

Siddiquee, S. and A. En-Nouaary (2006). Two architectures for testing distributed real-time systems. Volume 2, pp. 3388–3393.

Suliman, D., B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes, and R. Malaka (2006). The MORABIT Approach to Runtime Component Testing. In *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '06, Washington, DC, USA, pp. 171–176. IEEE Computer Society.

Tarhini, A. and H. Fouchal (2005). Conformance testing of real-time component based systems. In *ISSADS*, pp. 167–181.

Wall, A. (2000). Software architectures for real-time systems. Technical report.

Wang, Y., G. King, D. Patel, S. Patel, and A. Dorling (1999). On coping with real-time software dynamic inconsistency by built-in tests. *Ann. Softw. Eng. 7*(1-4), 283–296.

Résumé

Ce travail se concentre sur le test des systèmes temps-réels distribués dans le cas où leur configuration évolue dynamiquement, appelés aussi systèmes adaptables. Dans ce contexte, le test durant l'exécution est émergé comme une nouvelle solution pour valider un tel système. Afin de réduire l'effort, le coût et le temps de test, nous appliquons la technique d'analyse de dépendances ayant pour rôle d'identifier les parties du système affectées par la reconfiguration dynamique. De plus, nous proposons une architecture de test distribuée évolutive et flexible formée de deux types de testeurs : testeurs niveau composant et testeurs niveau composition de composants. Ces testeurs exécutent des tests unitaires (respectivement d'intégration) sur les composants (respectivement les compositions de composants) affectés par les actions de reconfiguration. Un exemple illustratif décrivant les interactions entre ces testeurs quand deux scénarios de reconfiguration auront lieu est présenté.