Towards a Runtime Standard-based Testing Framework for Dynamic Distributed Information Systems

Moez Krichen[1,3], Roobaea Alroobaea[2] and Mariam Lahami[3]
[1]*Faculty of CSIT, Al-Baha University, Saudi Arabia*
[2]*College of CIT, Taif University, Saudi Arabia*
[3]*ReDCAD Laboratory, University of Sfax, Tunisia*

Keywords: Distributed, Information Systems, Runtime, Dynamic, Adaptable, TTCN-3, Testing, Isolation, Generation, Minimization, Classification.

Abstract: In this work, we are interested in testing dynamic distributed information systems. That is we consider a decentralized information system which can evolve over time. For this purpose we propose a runtime standard-based test execution platform. The latter is built upon the normalized TTCN-3 specification and implementation testing language. The proposed platform ensures execution of tests cases at runtime. Moreover it considers both structural and behavioral adaptations of the system under test. In addition, it is equipped with a test isolation layer that minimizes the risk of interference between business and testing processes. The platform also generates a minimal subset of test scenarios to execute after each adaptation. Finally, it proposes an optimal strategy to place the TTCN-3 test components among the system execution nodes.

1 INTRODUCTION

Nowadays, information systems are steadily gaining importance as a major ingredient for a wide range of modern technologies and computerized services. More precisely we are interested in distributed information systems (Grusho et al., 2017; Yesikov et al., 2017) which correspond to decentralized systems made of sets of physical devices and software components. Moreover we pay a particular attention to dynamic distributed information systems which are systems that can evolve during execution time.

The possible changes the considered system may encounter could be either structural or behavioral. On the first hand, structural changes correspond to the addition or deletion of physical nodes or links between nodes. On the other hand, behavioral changes correspond to the addition, deletion or update of the software components installed on the physical nodes of the system.

It is worth noting that these two types of changes may happen while the system is running. Consequently this may lead to the apparition of new risks, bugs and problems for the considered distributed information system either locally (on some limited parts of the system) or globally (on the whole behavior of the system).

In both cases this is considered as a dangerous and critical situation. Thus very urgent and efficient measures must be taken in order to guarantee the correctness and the safety of the distributed information system we are dealing with.

To remedy this, we adopt in this work an online testing approach which can be applied at *runtime*. For that, we propose a standard-based testing platform for dynamic distributed information systems. The proposed platform uses the TTCN3 standard and takes into account both structural and behavioral adaptations. Moreover, it is equipped with a test isolation layer which minimizes the risk of interference between testing and business processes. We also compute a minimal subset of test cases to run and efficiently distribute them among the execution nodes.

The remainder of this paper is organized as follows. Section 2 is dedicated for Runtime Testing for Structural Adaptations. In Section 3, we propose techniques for Runtime Testing of Behavioral Adaptations. Finally, Section 4 summarizes the main contributions presented in this paper and states some directions for future work.

Krichen, M., Alroobaea, R. and Lahami, M.
Towards a Runtime Standard-based Testing Framework for Dynamic Distributed Information Systems.
DOI: 10.5220/0007772101210129
In *Proceedings of the 21st International Conference on Enterprise Information Systems (ICEIS 2019)*, pages 121-129
ISBN: 978-989-758-372-8
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 TESTING FOR STRUCTURAL ADAPTATIONS

The process depicted in Figure 1 shows the different steps to fulfill in order to execute runtime tests when structural reconfiguration actions happen:

Figure 1: The different steps for structural adaptations validation of the system under test.

- **Dependency Analysis:** This step focuses on determining the affected components of the system under test by a structural reconfiguration action.
- **Test Case Selection:** This step allows to identify the set of test cases to choose from the *Test Case Repository* and which will be applied on the set of components identified during the previous step.
- **Test Component Placement:** This step allows to distribute test components over the execution nodes while taking into account specific constraints.
- **Test Isolation and Execution:** This step allows to take into account isolation aspects and to execute the test cases selected during step 2.

More details about these different steps are given in the next sections.

2.1 Dependency Analysis

Our goal here is to reduce the cost of the test activity with respect to both to time and resources. For that purpose, after each dynamic evolution of the system we determine the subset of components of the system which were impacted by the considered dynamic evolution. This allows us not to execute all the available test cases at runtime. Rather we choose a minimal subset of tests (corresponding to the modified parts of the system) and then we re-execute them. Our technique is based on the use of a dependency analysis algorithm. The latter is used in several software engineering branches (e.g., testing (Li et al., 2005), maintenance and evolution (Qu et al., 2010), etc.).

In (Alhazbi and Jantan, 2007) the authors define dependency as "the reliance of a component on other(s) to support a specific functionality". So It can be seen as a binary relation between components. A component C_1 is said to be an *antecedent* to component C_2 if its functionalities or data are exploited by C_2. Moreover in this case, B is said to be *dependent* on C_1. For a such situation we use the following notation $C_2 \rightarrow C_1$.

The authors of (Larsson and Crnkovic, 2001) define the relation $\rightarrow$ in a formal fashion as follows. Let $\mathcal{N}odes$ be the set of components of the considered system. The set of dependencies of this system is:

$$\mathcal{D}ep = \left\{(C_p, C_q) : C_p, C_q \in \mathcal{N}odes \wedge C_p \rightarrow C_q\right\}.$$

In this manner, the current configuration of the system is made of two elements namely the set of components and the set of corresponding dependencies

$$\mathcal{C}onf = (\mathcal{N}odes, \mathcal{D}ep).$$

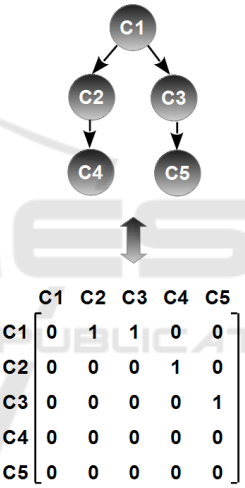

Figure 2: An example of a 5-nodes dependency graph and its corresponding 5x5 dependency matrix.

The configuration of the system can be presented either by a *Component Dependency Graph* or an equivalent *Component Dependency Matrix*. In Figure 2 we give an example of such a dependency graph and its equivalent dependency matrix.

At the beginning, the dependency graph stores only the direct dependencies between the different components of the system. Then in a second step indirect dependencies are calculated by means of transitive closure algorithm (Ioannidis and Rantakrishnan, 1988) applied on the initial graph. For instance in the example given by Figure 2 there is no direct dependency between the two nodes $C1$ and $C4$ however an indirect dependency exist between them through the node $C2$. Similarly an indirect dependency exist between $C1$ and $C5$ through the node $C3$.

2.2 Test Case Selection

When dynamic adaptations occur, we need to consider two types of test cases. The first type corresponds to *unit tests* which allow to check the correctness of the components which were affected at a local level. Besides, the second type corresponds to *integration tests* which are necessary to guarantee compatibility and correct interaction between the different components of the system which were influenced by the dynamic adaptation.

As shown in Figure 3, this step consists in identifying a subset of test cases from an already existing test suite in order to test the affected parts of the system. For that purpose several regression test selection methods have been adopted in the literature. The proposed methods are mostly based on dependency analysis techniques (Rothermel and Harrold, 1996).

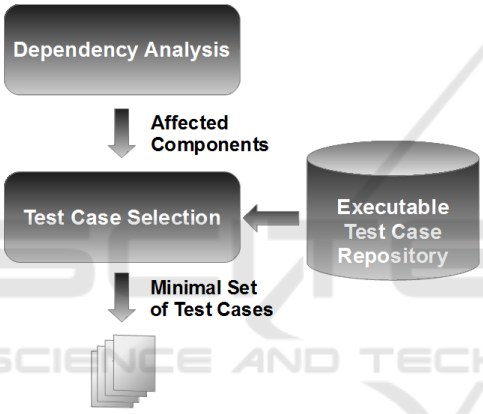

Figure 3: Illustration of the test selection phase.

Since we opted for a standard-based test execution platform, our tests will be written in the TTCN-3 language and run by TTCN-3 test components. More precisely two types of TTCN-3 components are needed namely *Main Test Components* (MTC) and *Parallel Test Components* (PTC). As illustrated in Figure 4, an MTC can be charged of the execution of a unit test for a specific component of the system under test (Figure 4.a). Otherwise it can collaborate with a set of PTCs in order to execute some integration tests corresponding to a set of interacting components of the system (Figure 4.b). In the next section we explain how the placement of these different test components is done.

2.3 Test Component Placement

In this section, we explain how to compute an adequate plan to distribute test cases over the different nodes of the execution environment. For this purpose we consider both connectivity and resource constraints of the different components of the system.

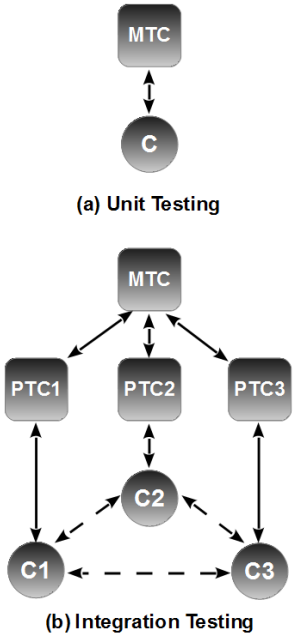

Figure 4: TTCN-3 components used to execute unit and integration test cases.

First the connectivity constraint corresponds to the fact that a given TTCN-3 test component needs to have a connection (either direct or indirect) with all components of the system it is in charge of testing.

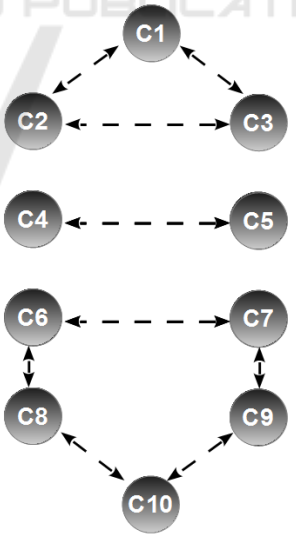

Figure 5: An example of a system under test made of ten components.

For instance the system shown in Figure 5 is made of ten components. These components can be divided into three subsets according to connectivity namely $\{C1, C2, C3\}$, $\{C4, C5\}$ and $\{C6, C7, C8, C9, C10\}$.

Consequently at least three test components are needed in order to test this system (i.e., a TTCN-3 test component for each of the previously defined subsets of the system nodes).

The second constraint to take into account at this level deals with the availability of resources required by test components in order to execute the set of selected test cases. That is each node of the execution environment which is hosting a particular test component must have enough resources to execute the test cases attributed to this test component. Among these resources we may consider for instance the CPU load, the memory occupation, the storage capacities, etc.

Computing an adequate test placement strategy is done by fitting the previous constraints. This procedure may be considered as a *Constraint Satisfaction Problem* (CSP) (Ghédira and Dubuisson, 2013). More precisely the problem in hand can be instantiated as a particular variant of the Knapsack Problem (Martello and Toth, 1990; Kellerer et al., 2004), called *Multiple Multidimensional Knapsack Problem* (MMKP) (Lahami et al., 2012b).

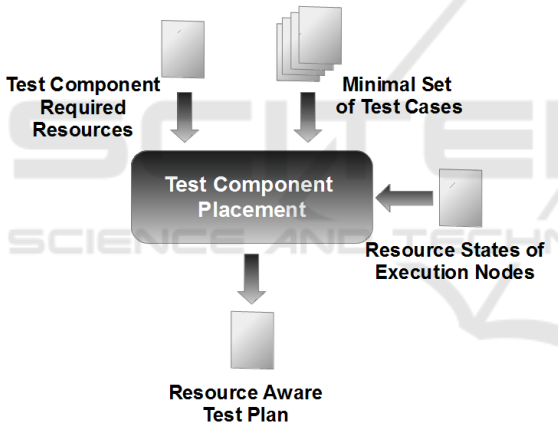

Figure 6: Illustration of the TTCN-3 Test Component Placement Module.

As presented in Figure 6, the test component placement takes as inputs the minimal set of test cases (generated during the test selection phase), a description of the amount of resources needed by the test components and finally a description of the state of available resources in each node of the execution platform. As an output this module generates a resource aware test plan.

2.4 Test Isolation and Execution

As already mentioned our work is based on the TTCN-3 standard. More precisely we are inspired by the work of (Lahami et al., 2012a; Lahami et al., 2016). Next we recall the main constituents of the TTCN-3 reference architecture as shown in Figure 7:

- **Test Management (TM):** starts and stops tests, provides parameters and manages the whole test process;
- **Test Logging (TL):** handles all log events;
- **TTCN-3 Executable (TE):** executes the compiled TTCN-3 code;
- **Component Handling (CH):** distributes parallel test components and ensures communication between them;
- **Coding and Decoding (CD):** encodes and decodes data exchanged with the TE;
- **System Adapter (SA):** adapts the communication with the system under test;
- Platform Adapter (PA): implements external functions.

In Figure 7, the abbreviation TCI stands for *TTCN-3 Control Interface* and the abbreviation TRI stands for *TTCN-3 Control Interface*.

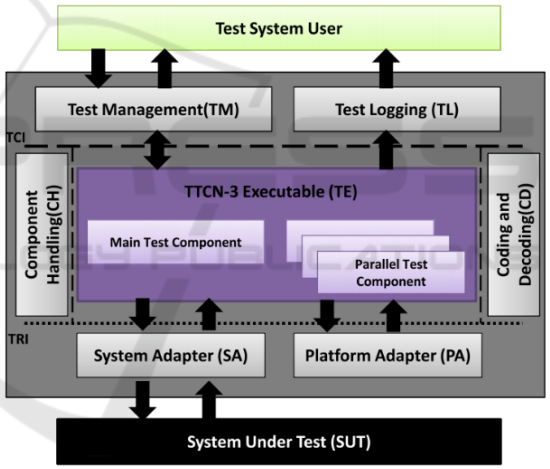

Figure 7: TTCN-3 Reference Architecture (Lahami et al., 2016).

As already mentioned, we need to consider some isolation operations during the test execution phase in order to avoid any possible interference between business behaviors and test behaviors. So inspired by the work of (Lahami et al., 2016), we intend to extend the standard TTCN-3 architecture with a *test isolation component* (Figure 8). This new component is connected with both the system adapter (SA) and the system under test. It allows to choose the appropriate isolation strategy for each constituent under test depending on the nature of that constituent. Next, we briefly describe five possible isolation strategies:

- **The BIT-based Strategy:** corresponds to a built-in-testing approach where we assume that the

SUT components are initially equipped with interfaces which accept both business inputs and test inputs simultaneously.

- **The Aspect-based Strategy:** which is inspired from Aspect-Oriented Programming (AOP) paradigm (Kienzle et al., 2010; Kiczales et al., 2001) and consists in equipping each SUT component with an aspect providing testing facilities.
- **The Tagging-based Strategy:** consists in tagging test inputs with a special flag to differentiate them from business inputs.
- **The Cloning-based Strategy:** consists in creating a copy of the component under test and then testing the created clone instead of the original component.
- **The Blocking-based Strategy:** consists in preventing temporarily the component under test from receiving any requests from its environment except from the test components to which it is connected.

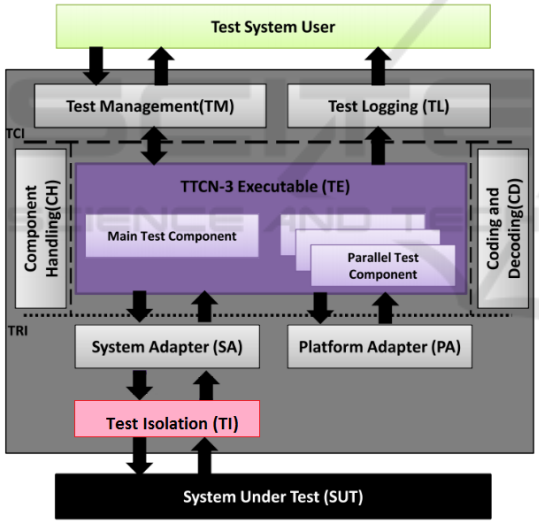

Figure 8: TTCN-3 Architecture Extended with an Isolation Component (Lahami et al., 2016).

3 TESTING OF BEHAVIORAL ADAPTATIONS

Now we move to the second part of this work which deals with runtime testing of distributed information systems after behavioral adaptations. For this purpose we adopt a formal approach (Krichen, 2010; Krichen and Tripakis, 2009) based on the use of the model of *timed automata* (Alur and Dill, 1994). That is we assume the our system is modelled as a simple timed automaton or a product of timed automata. Moreover we assume the considered model may evolve either partially or entirely after the occurrence of a dynamic behavioral adaptation. Consequently there is a need to update the set of available test cases either by creating new tests or modifying old ones. For that reason we need to take advantage from both selective regression testing and model-based testing techniques.

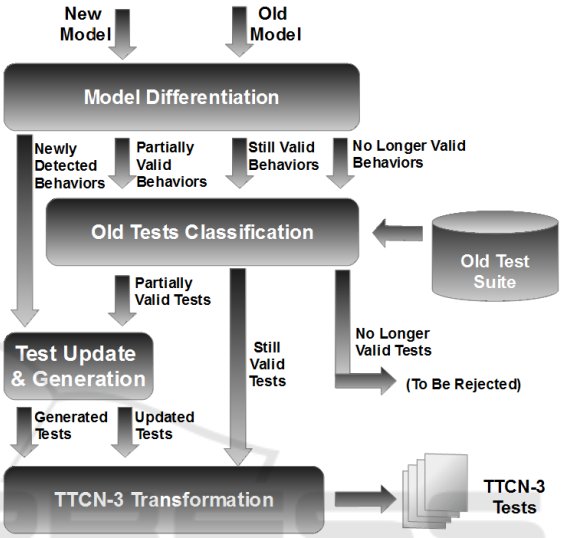

Figure 9: The different steps for behavioral adaptations validation of the system under test.

In order to reach this goal, we adopt a four-step approach as illustrated in Figure 9. The four considered steps are the following:

- **Model Differentiation:** consists in comparing the old and the new model of the distributed information system we have in hand in order to detect similarities and differences between them;
- **Old Tests Classification:** consists in sorting the old tests and dividing them into three groups (1. Tests which need to be removed because they are no longer valid, 2. Tests which are partially valid and need to be updated and finally 3. Tests which are still valid and do not need any update);
- **Test Update and Generation:** consists in updating the partially valid tests detected in the previous step and generating new test cases covering the new behaviors appearing in the new model of the system;
- **TTCN-3 Transformation:** consists in concretizing the set of obtained tests by translating them from an abstract level into a TTC-3 representation.

More details about these different steps are given in the next subsections. However before that we start by giving a brief description of the formalism used to model distributed information systems, namely UPPAALL timed automata.

3.1 UPPAAL Timed Automata

In order to specify the behavioral models of evolved systems, *Timed Automata* (TA) are chosen for the reason that they correspond to a widespread formalism usually used for modeling behaviors of critical and real-time systems. More precisely, we opt for the particular UPPAAL style (Behrmann et al., 2004) of timed automata because UPPAAL is a well-established verification tool. It is made up of a system editor that allows users to edit easily timed automata, a simulator that visualizes the possible dynamic execution of a given system and a verifier that is charged with verifying a given model w.r.t. a formally expressed requirement specification. Within UPPAAL timed automata, a system is modeled as a network of timed automata, called processes. A timed automaton, is an extended finite-state machine equipped with a set of clock-variables that track the progress of time and that can guard when transitions are allowed.

Let $\mathcal{C}l$ be a set of continuous variables called clocks, and $\mathcal{A}\text{ct} = Inp \cup Out\}$ where Inp is a set of inputs and Out a set of outputs. $\mathcal{G}d(\mathcal{C}l)$ is defined as the set of guards on the clocks $\mathcal{C}l$ being conjunctions of constraints of the form $c\ op\ n$, where $c \in \mathcal{C}l$, $n \in \mathbb{N}$, and $op \in \{\leqslant, \leq, =, \geq, \geqslant\}$. Moreover, let $\mathcal{U}p(\mathcal{C}l)$ denote the set of updates of clocks of the form $c := n$.

A timed automaton over $(\mathcal{A}\text{ct}, \mathcal{C}l)$ is a tuple $(Loc, l_0, \mathcal{A}\text{ct}, \mathcal{C}l, Inv, Ed)$, where :

- Loc is the set of locations;
- $l_0 \in Loc$ is the initial location;
- $Inv : Loc \longmapsto \mathcal{G}d(\mathcal{C}l)$ a function which an invariant to each location;
- Ed is the set of edges such that $Ed \subseteq Loc \times \mathcal{G}d(\mathcal{C}l) \times \mathcal{A}\text{ct}_\tau \times \mathcal{U}p(\mathcal{C}l) \times Loc$.

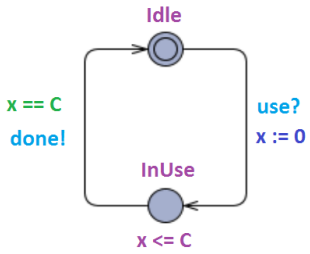

Figure 10: A simple UPAALL timed automaton.

A simple example of a UPPAALL timed automaton is shown in Figure 10. It has:

- Two locations: $Idle$ and $InUSe$;
- One input action: $use?$;
- One output action $done!$;
- One clock (x).

The system is initially occupying the location $Idle$. The clock x is reset to 0 when the input action $use?$ takes place. The system is allowed to stay at most C times unit at the location $InUse$. Finally when the clock x reaches the value C the output action $done!$ is produced and the system goes back to its initial location. Then the same cycle can start again and again.

3.2 Model Differentiation

As already mentioned the goal of this step is to compare the old and new models of the system under test. That is to find differences and similarities between them. This allows us to avoid generating the whole set of test cases from scratch after each behavioral adaptation of the system. For this purpose we need to split the behaviors described by the new model of the system into four groups (as illustrated in Figure 9):

- **Newly Detected Behaviors:** correspond to the set of behaviors present in the new model of the system but not in the old one;
- **Partially Valid Behaviors:** correspond to the set of behaviors that bear a high level of similarity with some old behaviors of the system but are not completely identical to them;
- **Still Valid Behaviors:** correspond to the set of behaviors which are in common between the two versions of the model of the system;
- **No Longer Valid Behaviors:** correspond to the set of behaviors present in the old model of the system but not in the new one and which therefore must be rejected.

To achieve this goal, we need to take advantage from different works present in the literature. For instance, (Pilskalns et al., 2006) presented a technique which allows to reduce the complexity of identifying the modification impact from different UML diagrams. The authors consider different design changes which are classified according to whether they create, modify, or delete elements in the diagram.

In this same context, (Chen et al., 2007) propose a regression technique based on the use of the model of *Extended Finite State Machine*. This model is used to identify the effects of three types of modifications: modification, addition or deletion of a transition.

3.3 Old Tests Classification

Once the computation of the different sets of behaviors is achieved as explained in the previous step, the next task to do consists in splitting the set of old available test cases accordingly into three subsets of tests (see Figure 9):

- **Partially Valid Tests:** correspond to the set of partially valid behaviors of the new model of the system;
- **Still Valid Tests:** correspond to the set of still valid behaviors of the new model;
- **No Longer Valid Tests:** correspond to the set of no longer valid behaviors present in the old model of the system and which must be rejected.

At this level too we need to take advantage from existing contributions in the literature. For example, (Briand et al., 2009) introduce a UML-based regression test selection strategy. The latter allows to classify tests issued from the initial behavioral models as obsolete, reusable and re-testable tests.

Similarly, (Fourneret et al., 2011) proposed a technique which selects reusable tests, discards obsolete ones and generates only new ones.

Moreover in (Korel et al., 2005), the authors split the set of old tests into high and low priority subsets. A high priority is attributed to tests that manipulate modified transitions and a low priority to tests that do not include any modified transition.

3.4 Test Update and Generation

As illustrated in Figure 9, we need to update partially valid tests identified during the test classification step and to derive new test cases from the newly detected behaviors identified during the model differentiation step. For this purpose we need to use techniques borrowed from model-based testing approaches (Krichen and Tripakis, 2009; Krichen, 2012).

More precisely the technique we intend to adopt consists in using a particular type of timed automata called *observer automata* (Blom et al., 2005). These automata allow to model the specific behaviors we are interested in testing. In our case the behaviors to model are the newly detected behaviors discovered during the differentiation step.

The main steps to follow are the following:

1. Transforming the desired behavior into an observer automaton;
2. Computing the product of the observer automaton and the new timed automaton model of the system under test;
3. Deriving tests cases from the obtained timed-automaton product using classical test generation techniques.

The previously mentioned steps can be performed using the test generation tool UPPAAL CO$\sqrt{}$ER (Hessel and Pettersson, 2007) which supports the notion of observers and test case generation.

3.5 TTCN-3 Transformation

This step consists in defining a set of rules to derive TTCN-3 tests from abstract test cases obtained during the previous steps. Our transformation algorithm will be inspired by the following works (Axel Rennoch and Schieferdecker, 2016; Lahami et al., 2012a; Hochberger and Liskowsky, 2006; Ebner, 2004).

In table 1 we give some examples of the rules to use to translate abstract test cases into concrete TTCN-3 tests. These rules are briefly explained below:

- **R1:** consists in generating a new TTCN-3 module for each considered abstract test suite;
- **R2:** consists in transforming each abstract test sequence into a concrete TTCN-3 test case;
- **R3:** consists in associating a TTCN-3 timer with each abstract timed behavior;
- **R4:** consists in transforming each abstract test sequence (with the form input-delay-output) into a concrete TTCN-3 function;
- **R5:** consists in transforming the abstract channels (declared in the UPPAAL XML file) into concrete TTCN-3 templates.

Table 1: TTCN-3 Transformation Rules.

R#	Abstract Concepts	TTCN-3 Concepts
R1	Test Suite	TTCN-3 Module
R2	Single Trace	TTCN-3 Test Case
R3	Timed Behavior	TTCN-3 Timer
R4	Test Sequence	TTCN-3 Function
R5	Channel	TTCN-3 Template

4 CONCLUSION

In this article, we adopted a runtime testing approach to validate and test dynamic distributed information systems. For this purpose, we proposed a standard-based and resource aware test execution platform which allows to deal efficiently with both structural and behavioral adaptations. On the one hand structural adaptations correspond to the addition or deletion of nodes and/or connections between the nodes

of the system under test. On the other hand behavioral adaptations correspond to the addition, deletion or update of the software components or functionalities installed on the physical devices of the considered system under test.

Firstly, Our platform is standard-based since it is built upon the TTCN-3 specification and execution testing language. Indeed we adopted the generic standard TTCN-3 architecture extended with an isolation layer which allows to avoid possible risks of interference between testing and business processes. To reach this goal, five possible isolation strategies may be adopted at this level namely: 1. BIT-based, 2. aspect-based, 3. tagging-based, 4. cloning-based, and finally 5. blocking-based.

Secondly, our platform is resource aware since it ensures test selection and optimal placement of test components among physical nodes of the system under test. The test selection phase consists in identifying a subset of tests scenarios from an already existing repository in order to test the affected parts of the system. Besides, the placement phase consists in computing an appropriate plan to assign test cases to the different nodes of the execution environment using adequate optimization techniques.

Regarding behavioral adaptations, our methodology comprises four steps. The first step corresponds to the differentiation between the old and the new models of the system in order to identify similarities and differences between them. The second step consists in sorting the old tests and splitting them into three suitable sets. The third step lies in updating a subset of old tests and in generating a set of new test cases. Finally, the fourth step consists in translating the set of obtained tests scenarios from an abstract representation into the TTCN-3 language.

Our work is at its beginning and many efforts are still needed on both theoretical and practical aspects in order to achieve the different goals mentioned in this article:

- We need to build an appropriate model for the considered system under test. At this level we need to choose an optimal level of abstraction for this model. That is the latter should be neither too small nor too big. Being too small may cause the loss of important details of the system and being too big may lead to the famous state explosion problem which may make our approach completely infeasible.
- We also need to consider some sophisticated heuristics to solve the test component placement problem. In fact since we are adopting a runtime testing approach then the time to find an acceptable placement plan should be enough short in order to detect anomalies at the right moment.
- Moreover we need to propose efficient technical solutions to implement the different isolation strategies mentioned previously in this article. In the same manner we should develop and install adequately the different TTCN-3 test components upon the physical devices of the system in order to guarantee the success of the whole approach.
- Finally we may combine load and functional testing aspects as done in (Krichen et al., 2018; Maâlej et al., 2013; Maâlej et al., 2012b; Maâlej et al., 2012a).

REFERENCES

Alhazbi, S. and Jantan, A. (2007). Dependencies Management in Dynamically Updateable Component-Based Systems. *Journal of Computer Science*, 3(7):499–505.

Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

Axel Rennoch, Claude Desroches, T. V. and Schieferdecker, I. (2016). TTCN-3 Quick Reference Card.

Behrmann, G., David, A., and Larsen, K. (2004). A tutorial on uppaal. In Bernardo, M. and Corradini, F., editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *LNCS*, pages 200–237. Springer Verlag.

Blom, J., Hessel, A., Jonsson, B., and Pettersson, P. (2005). Specifying and Generating Test Cases Using Observer Automata. In *Proceeding of the 5th International Workshop on Formal Approaches to Software Testing (FATES'05)*, pages 125–139.

Briand, L. C., Labiche, Y., and He, S. (2009). Automating Regression Test Selection Based on UML Designs. *Information & Software Technology*, 51(1):16–30.

Chen, Y., Probert, R. L., and Ural, H. (2007). Model-based Regression Test Suite Generation Using Dependence Analysis. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing (A-MOST'07)*, pages 54–62.

Ebner, M. (2004). TTCN-3 Test Case Generation from Message Sequence Charts. In *Proceeding of the Workshop on Integrated-reliability with Telecommunications and UML Languages (WITUL'04)*.

Fourneret, E., Bouquet, F., Dadeau, F., and Debricon, S. (2011). Selective Test Generation Method for Evolving Critical Systems. In *Proceedings of the 2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11)*, pages 125–134.

Ghédira, K. and Dubuisson, B. (2013). *Constraint Satisfaction Problems*, chapter Foundations of CSP, pages 1–28. John Wiley & Sons, Inc.

Grusho, A. A., Grusho, N. A., and Timonina, E. E. (2017). Information security architecture synthesis in dis-

tributed information computation systems. *Automatic Control and Computer Sciences*, 51(8):799–804.

Hessel, A. and Pettersson, P. (2007). CO$\surd$ER A Real-Time Test Case Generation Tool. In *Proceeding of the 7th International Workshop on Formal Approaches to Testing of Software (FATES'07)*.

Hochberger, C. and Liskowsky, R., editors (2006). *Informatik 2006 - Informatik für Menschen, Band 2, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2.-6. Oktober 2006 in Dresden*, volume 94 of *LNI*. GI.

Ioannidis, Y. E. and Rantakrishnan, R. (1988). Efficient Transitive Closure Algorithms. In *Proceedings of the 14th International Conference on Very Large Databases (VLDB'88)*.

Kellerer, H., Pferschy, U., and Pisinger, D. (2004). *Knapsack Problems*. Springer, Berlin, Germany.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327–353.

Kienzle, J., Guelfi, N., and Mustafiz, S. (2010). *Transactions on Aspect-Oriented Software Development VII: A Common Case Study for Aspect-Oriented Modeling*, chapter Crisis Management Systems: A Case Study for Aspect-Oriented Modeling, pages 1–22. Springer Berlin Heidelberg.

Korel, B., Tahat, L., and Harman, M. (2005). Test Prioritization Using System Models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance(ICSM'05)*, pages 559–568.

Krichen, M. (2010). A Formal Framework for Conformance Testing of Distributed Real-Time Systems. In *Proceedings of the 14th International Conference On Principles Of Distributed Systems, (OPODIS'10)*.

Krichen, M. (2012). A formal framework for black-box conformance testing of distributed real-time systems. *IJCCBS*, 3(1/2):26–43.

Krichen, M., Maâlej, A. J., and Lahami, M. (2018). A model-based approach to combine conformance and load tests: an ehealth case study. *IJCCBS*, 8(3/4):282–310.

Krichen, M. and Tripakis, S. (2009). Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304.

Lahami, M., Fakhfakh, F., Krichen, M., and Jmaïel, M. (2012a). Towards a TTCN-3 Test System for Runtime Testing of Adaptable and Distributed Systems. In *Proceedings of the 24th IFIP WG 6.1 International Conference Testing Software and Systems (ICTSS'12)*, pages 71–86.

Lahami, M., Krichen, M., Bouchakwa, M., and Jmaiel, M. (2012b). Using knapsack problem model to design a resource aware test architecture for adaptable and distributed systems. In *Testing Software and Systems - 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings*, pages 103–118.

Lahami, M., Krichen, M., and Jmaïel, M. (2016). Safe and Efficient Runtime Testing Framework Applied in Dynamic and Distributed Systems. *Science of Computer Programming (SCP)*, 122(C):1–28.

Larsson, M. and Crnkovic, I. (2001). Configuration Management for Component-Based Systems. In *Proceeding of the 10th International Workshop on Software configuration Management (SCM'01)*.

Li, B., Zhou, Y., Wang, Y., and Mo, J. (2005). Matrix-based Component Dependence Representation and Its Applications in Software Quality Assurance. *ACM SIGPLAN Notices*, 40(11):29–36.

Maâlej, A. J., Hamza, M., Krichen, M., and Jmaiel, M. (2013). Automated significant load testing for WS-BPEL compositions. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, pages 144–153.

Maâlej, A. J., Krichen, M., and Jmaiel, M. (2012a). Conformance testing of WS-BPEL compositions under various load conditions. In *36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, page 371.

Maâlej, A. J., Krichen, M., and Jmaiel, M. (2012b). Model-based conformance testing of WS-BPEL compositions. In *36th Annual IEEE Computer Software and Applications Conference Workshops, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, pages 452–457.

Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA.

Pilskalns, O., Uyan, G., and Andrews, A. (2006). Regression Testing UML Designs. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 254–264.

Qu, B., Liu, Q., and Lu, Y. (2010). A Framework for Dynamic Analysis Dependency in Component-Based System. In *the 2nd International Conference on Computer Engineering and Technology (ICCET'10)*, pages 250–254.

Rothermel, G. and Harrold, M. (1996). Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551.

Yesikov, D., Ivutin, A., Larkin, E., and Kotov, V. (2017). Multi-agent approach for distributed information systems reliability prediction. *Procedia Computer Science*, 103:416 – 420. XII International Symposium Intelligent Systems 2016, INTELS 2016, 5-7 October 2016, Moscow, Russia.