Contributions to Model-Based Testing of Dynamic and Distributed Real-Time Systems

Moez Krichen

The Minister of Higher Education and Scientific Research

University of Sfax
National Engineering School of Sfax

Doctoral School Sciences and Technologies
HDR Thesis
Computer System Engineering
Order N°:

THESIS

Presented at

National Engineering School of Sfax

to obtain the title of

HDR IN COMPUTER SCIENCES

Computer System Engineering

by

Moez KRICHEN

Contributions to Model-Based Testing of Dynamic and Distributed Real-Time Systems

Defended on 15 August 2018 in front of the jury composed of:

Prof. Mohamed Jmaiel	(University of Sfax, Tunisia)	Chair
Prof. Kamel Barkaoui	(CEDRIC-CNAM, France)	Reviewer
Prof. Adel Mahfoudhi	(University of Sfax, Tunisia)	Reviewer
Prof. Wassim Jaziri	(University of Sfax, Tunisia)	Examiner
A.Pr. Mahdi Khemakhem	(University of Sfax, Tunisia)	Examiner

Abstract

In this dissertation we report on our main research contributions dealing with Model-Based Testing of Dynamic and Distributed Real-Time Systems, performed during the last ten years.

Our first contribution deals with testing techniques for distributed and dynamically adaptable systems. In this context, we propose a standard-based test execution platform which affords a platform-independent test system for isolating and executing runtime tests. This platform uses the TTCN3 standard and considers both structural and behavioral adaptations. Moreover, our platform is equipped with a test isolation layer that reduces the risk of interference between testing processes and business processes. Besides, we compute a minimal subset of test cases to run and efficiently distribute them among the execution nodes while respecting resource and connectivity constraints. In addition, we validate the proposed techniques on two case studies, one in the healthcare domain and the other one in the fleet management domain.

Our second contribution consists in proposing a model-based framework to combine Load and Functional Tests. This framework is based on the model of extended timed automata with inputs/ouputs and shared integer variables. We present different modelling techniques aspects and we illustrate them by means of a case study. Moreover, we study BPEL compositions behaviors under various load conditions using the proposed framework. We introduce a taxonomy of the detected problems and we illustrate how test verdicts are assigned. Besides, we validate our approach using a Travel Agency case study. Furthermore, we consider several mutants of the corresponding BPEL process and we test them using our tool.

Our third contribution consists in introducing a set of formal techniques for the determinization and off-line test selection for timed automata with inputs and outputs. With this respect, we propose a game-based approach between two players for the determinization of a given timed automaton and some fixed resources. Moreover, we present a complete formalization for the automatic off-line generation of test cases from non-deterministic timed automata with inputs and outputs. We also define a selection technique of test cases with expressive test purposes. Test cases are generated using a symbolic co-reachability analysis of the observable behaviors of the specification guided by the test purpose which is in turn defined as a special timed automaton.

Finally we report on two ongoing works. The first one deals with a model-based approach for security testing of Internet of Things applications. The second one deals with providing a scalable test execution platform providing testing facilities as a cloud service.

Contents

List of Figures

CHAPTER 1

General Introduction

1.1 Research Context and Motivation

In order to build and deliver quality assured software and avoid potential costs caused by unstable software, testing is a definitely essential step in software life cycle development. During the last few decades, very critical programming errors and accidents have been detected in different domains and in different corners of the world. Some of these errors were very dangerous and caused huge and dramatic human/financial/environmental damages.

A first example of critical software errors we cite is related to the medical field. From 1985 to 1987, at least four patients died as a direct result of a radiation overdose received from the medical radiation therapy device Therac-25. In fact, the victims received up to 100 times the required dose. The accident was the result of a bug in the software powering the Therac-25 device. A second example concerns the European rocket Ariane 5 explosion in 1996 just 37 seconds after launch. The explosion was the result of a wrong reuse of code from Ariane 4. The financial loss caused by this accident was estimated to be about $400 millions. A third example of software errors struck the very famous web service provider Google. This accident occurred in February 2009. Obviously, many other critical errors happened in many other fields. However, we restrict ourselves to the three previous introduced examples.

The important issue to emphasize here is that a good percentage of these errors could have been avoided by considering some more refined testing efforts. Yet, such efforts are still minimal in practice and the need for advanced testing solutions is still deep. Indeed, software companies are still not making enough efforts at this level.

Runtime Testing of Dynamically Adaptable and Distributed Systems: Nowadays, distributed component-based systems tend to evolve dynamically without stopping their execution. Known as *Dynamically Adaptable and Distributed Systems*, these systems are currently playing an important role in society's services. Indeed, the growing demand for such systems is obvious in several application domains such as crisis management (4), medical monitoring (5; 6), fleet management (7), etc. This demand is stressed by the complex, mobile and critical nature of these applications that also need to continue meeting their functional and non-functional requirements and to support advanced properties such as context awareness and mobility. Nevertheless, dynamic adaptations of component-based systems may generate new risks of bugs, unpredicted interactions (e.g., connections going down), unintended operation modes and performance degradation. This may cause system malfunctions and guide its execution to an unsafe state. Therefore, guaranteeing their high quality and their trustworthiness remains a crucial requirement to be considered. One of the most promising ways of testing dynamic systems is the use of an emerging technique, called *Runtime Testing*. In this work, we propose a standard-based test execution platform which affords a platform-independent test system for isolating and executing runtime tests. We also compute a minimal subset of test cases to run and efficiently distribute them among the execution nodes.

Combining Load and Functional Tests: Many systems ranging from e-commerce websites to telecommunications must support concurrent access by hundreds or thousands of users. In order to assure the quality of these systems, load testing is a required testing process in addition to conventional functional testing procedures, which focus on testing a system based on a small number of users (8). In fact, load testing is one of the testing types with high importance. It is usually accompanied by performance monitoring of the hosting environment. Typically, industry software testing practice is to separate load testing from functional testing. Different teams with different expertise and skills execute their testing at different times, and each team evaluates the results against its own criteria. It is exceptional to get the two testing types together and to evaluate load test results for functional correctness or incorporate sustained load in the functional testing. In this work, we propose a formal model-based framework to combine functional and load tests. Moreover, we study BPEL (Business Process Execution Language) compositions behaviors under various load conditions using the proposed framework.

Determinization of Timed Automata: Timed automata (TA), introduced in (9), form a usual model for the specification of real-time embedded systems. Essentially TAs are an extension of automata with guards and resets of continuous clocks. They are extensively used in the context of many validation problems such as verification, control synthesis or model-based testing.

Determinization is a key issue for several problems such as implementability, diagnosis or test generation, where the underlying analyses depend on the observable behavior. Our method combines techniques from (3) and (10) and improves those two approaches, despite their notable differences. The core principle is the construction of a finite turn-based safety game between two players, Spoiler and Determinizator, where Spoiler chooses an action and the region of its occurrence, while Determinizator chooses which clocks to reset. Our main result states that if Determinizator has a winning strategy, then it yields a deterministic timed automaton accepting exactly the same timed language as the initial automaton, otherwise it produces a deterministic over-approximation.

Off-Line Test Selection for Timed Automata: Conformance testing is the process of testing whether an implementation behaves correctly with respect to a specification. Implementations are considered as *black boxes*, the source code is unknown, only their interface with the environment is known and used to interact with the tester. In *formal model-based conformance testing* models are used to describe testing artifacts (specifications, implementations, test cases, ...), conformance is formally defined and test cases with verdicts are generated automatically. Then, the quality of testing may be characterized by properties of test cases which relate the verdicts of their executions with conformance (soundness). In this context, a very popular model is *timed automata with inputs and outputs* (TAIOs), a variant of *timed automata* (TAs) (11), in which observable actions are partitioned into inputs and outputs. We consider here partially observable and non-deterministic TAIOs with invariants for the modeling of urgency. In this work, we propose to generate test cases off-line for non-deterministic TAIOs, in the formal context of the tioco (2) conformance theory.

1.2 Contributions

The main research contributions presented in this dissertation are the following.

1. Testing Techniques for Distributed and Dynamically Adaptabe Systems:

   (a) We designed a standard-based test execution platform which affords a platform-independent test system for isolating and executing runtime tests. This platform uses the TTCN3 standard and considers both structural and behavioral adaptations. Moreover, our platform is equipped with a test isolation layer that reduces the risk of interference between testing processes and business processes.

   (b) We computed a minimal subset of test cases to run and efficiently distributed them among the execution nodes while respecting resource and connectivity constraints.

The minimal subset of test cases is obtained using a smart generation algorithm which keeps old tests cases which are still valid and replaces invalid ones by new generated or updated test cases.

(c) We validated the proposed techniques on two case studies, one in the healthcare domain and the other one in the fleet management domain. Through several experiments, we showed the efficiency of our tool in reducing the cost of runtime testing and we measure the overhead introduced in case of dynamic structural or behavioral adaptations.

2. A Model-Based Approach to Combine Load and Functional Tests:

   (a) We proposed a formal model-based framework to combine functional and load tests. The proposed framework is based on the model of extended timed automata with inputs/ouputs and shared integer variables. In addition, we presented different modelling issues illustrating some methodological aspects of our framework and we illustrated them by means of a case study.

   (b) We studied BPEL compositions behaviors under various load conditions using the proposed framework. We also proposed a taxonomy of the detected problems by our solution and we illustrated how test verdicts are assigned. Moreover, we validated our approach using a Travel Agency case study. We considered several mutants of the corresponding BPEL process and we tested them using our tool.

3. Formal Techniques for Determinization and Off-Line Test Selection for Timed Automata:

   (a) We proposed a game-based approach for the determinization of timed automata. For a given timed automaton $\mathcal{A}$ and some fixed resources, we build a finite turn-based safety game between two players Spoiler and Determinizator, such that any strategy for Determinizator yields a deterministic over-approximation of the language of $\mathcal{A}$ and any winning strategy provides a deterministic equivalent for $\mathcal{A}$.

   (b) We introduced a complete formalization for the automatic off-line generation of test cases from non-deterministic timed automata with inputs and outputs. We proposed an approximate determinization procedure and a selection technique of test cases with expressive test purposes. Test cases are generated using a symbolic co-reachability analysis of the observable behaviors of the specification guided by the test purpose.

1.3 Document Outline

The rest of this dissertation is structured in four parts as follows.

Part I: *Testing Distributed and Dynamically Adaptable Systems*

- **Chapter 2** presents the background material related to runtime testing of distributed and dynamically adaptable systems. Besides, it reports on related works.
- **Chapter 3** details the approach we propose to handle structural adaptations at runtime.
- **Chapter 4** introduces our proposal to handle behavioral adaptations at runtime.
- **Chapter 5** presents the prototype implementation of the RTF4ADS framework and reports on two case studies.

Part II: *Combining Load and Functional Tests*

- **Chapter 6** presents a classification of the load testing approaches and makes a comparative evaluation of them.
- **Chapter 7** propsoes a formal model-based framework to combine functional/load tests.
- **Chapter 8** reports on our study of BPEL compositions behaviors under various load conditions.

Part III: *Determinization and Off-Line Test Selection for Timed Automata*

- **Chapter 9** proposes a game-based approach for the determinization of timed automata..
- **Chapter 10** presents a formalization for the automatic off-line generation of test cases from non-deterministic timed automata.

Part IV: *Ongoing Works*

- **Chapter 11** reports on our ongoing work related to the model-based security testing of internet of things for smart cities.
- **Chapter 12** reports on our ongoing work dealing with providing a scalable test execution platform providing testing facilities as a cloud service.

Finally, **Chapter 13** summarizes the contributions and the obtained results and outlines several directions for future research.

Part I

Testing Distributed and Dynamically Adaptable Systems

CHAPTER 2

Background Materials and State of the Art

2.1 Introduction

This chapter is dedicated to present the background material and an overview on the sate of art related to the first contribution presented in this work. In Section 2.2, we start by giving the main characteristics of adaptable and distributed component-based systems and we discuss the challenges that we face after the occurrence of dynamic adaptations. Key concepts on software testing is outlined in Section 2.3. It includes software testing definition, test kinds, well-known test implementation techniques and test architectures. In Section 2.4, some testing techniques commonly used to validate modifications introduced in software systems are presented, namely regression and runtime testing. Section 2.5 outlines research work done mainly on test selection and test generation issues. The surveyed approaches in Section 2.6 are studied from different perspectives such as resource consumption, interference risks, test distribution, test execution and dynamic test evolution and generation. Finally, Section 2.7 summarizes the chapter.

2.2 Dynamically adaptable systems

2.2.1 Main characteristics

Dynamically adaptable systems (12) consist of a set of interconnected software components which are software modules that encapsulate a set of functions or data. Seen as black-boxes, components offer functionalities that are expressed by clearly defined interfaces. These interfaces are usually required to connect components for communication and to compose them in order

to provide complex functionalities (See Figure 2.1).

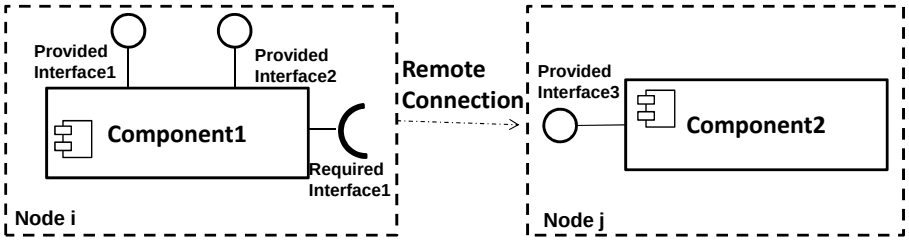


Figure 2.1: Distributed component-based architecture.

2.2.2 Dynamic adaptation: kinds and goals

Dynamic adaptation is defined in (13) as the ability to modify and extend a system while it is running. It can be either structural or behavioral. Figure 2.2 illustrates different kinds of structural reconfiguration actions.

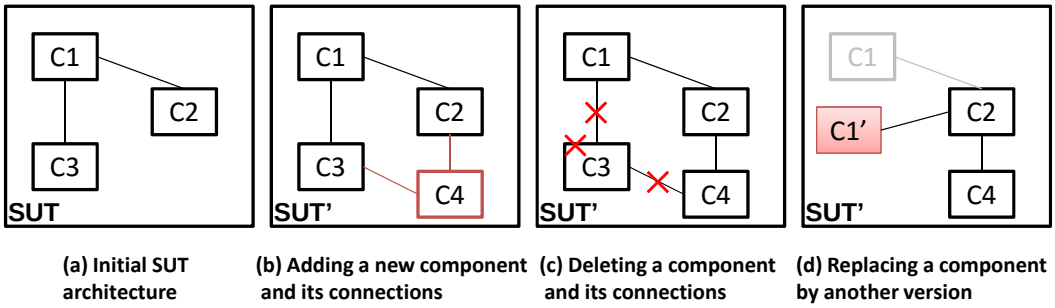


Figure 2.2: Basic structural reconfiguration actions.

For behavioral changes, the system behavior is modified by changing the implementation of its components or by changing its interfaces. Four major purposes of dynamic adaptation are defined in (14) :

- **Corrective adaptation:** removes the faulty behavior of a running component by replacing it with a new version that provides exactly the same functionality.
- **Extending adaptation:** extends the system by adding either new components or new functionalities.
- **Perfective adaptation:** aims to improve the system performance even if it runs correctly.
- **Adaptive adaptation:** allows adapting the system to a new running environment.

In the literature, several research approaches have been proposed to support the establishment of dynamic and distributed systems. Without loss of generality, we use in this work the OSGi (15) platform as a basis to build dynamic systems.

2.2.3 Challenges

By evolving dynamically the structure or/and the behavior of a distributed component-based system, several faults may arise at runtime. We distinguish:

- **Functional faults:** For instance, a defect at the software level can lead to an integration fault caused by interface or data format mismatches with its consumers.
- **Non-functional faults:** For instance, migrating a software component from one node to another can lead to performance degradation.

The failure of one component can trigger the failure of every component which is directly or indirectly linked to it. Also, all composite components that contain the faulty one may be subject to a failure. In the literature, two surveys address this issue (16; 17) and several *Validation and Verification* (V&V) techniques are proposed:

- **Model checking:** This technique exploits research done in the *Models at Run-Time* (M@RT) community (18; 17) in order to have an up-to-date representation of the evolved system. The latter is still a challenging issue since it is highly demanded to preserve coherence between runtime models and the running system (16; 19).
- **Monitoring and analysis of system executions:** Monitoring consists in observing passively system executions. The gathered data are then analyzed with the aim of detecting inconsistencies introduced after dynamic adaptions.
- **Software Testing:** To address the weakness imposed by the passive nature of monitoring, software runtime testing was introduced. It consists in stimulating the system with a set of test inputs and comparing the obtained outputs with a set of expected ones.

The latter technique is adopted in this work as one of the most effective V&V technique.

2.3 Software testing fundamentals

2.3.1 Levels and objectives

As shown in Figure 2.3, software testing is usually performed at different levels.

- **Unit testing:** in which individual units of the system are tested in isolation.
- **Integration testing:** in which the compatibility between components is checked.

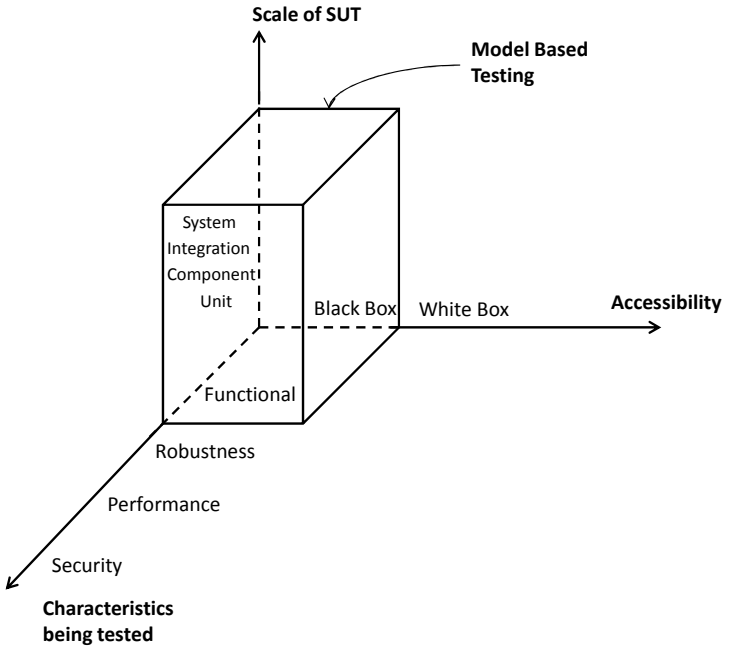


Figure 2.3: Different kinds of testing (1).

- **System testing:** in which the system formed from tested subsystems is tested as an entity.

Testing can be conducted to fulfill a specific objective. It can be used to verify different properties either functional or non-functional. For instance, test cases can be designed to validate whether the observed behavior of the tested software conforms to its specifications or not. This is mostly referred to in the literature as *Conformance testing*. Non-functional requirements, such as reliability, performance and security requirements, can be also validated by means of testing.

2.3.2 Test generation techniques

We distinguish mainly three categories:

- **Specification-based testing:** Formal SUT specifications (e.g., based on the Z specification language (20)) or object-oriented specifications (e.g., based on Object-Z notation (21)), are used for automatic derivation of functional test cases without requiring the knowledge of the internal structure of the program.
- **Model-Based Testing (MBT):** Test cases are derived from formal test models like *Unified Modeling Language* (UML) diagrams (22) and *Finite State Machine* (FSM) models (1).
- **Code-based testing:** Several approaches are proposed to extract test cases from the source code of a program. For instance, approaches in combinatorial testing (23), mutation testing (24), and symbolic execution (25) are seen as code-based test generation techniques.

2.3.3 Test implementation techniques

In the literature, we identify several test specification and test implementation techniques, including the *Java Unit* (JUnit) framework and the TTCN-3 standard (26).

- **JUnit:** It is designed for the Java programming language. JUnit exploits a set of assertion methods useful for writing self-checking tests. Compared to its old version (version 3), JUnit 4 makes use of annotations which provide more flexibility and simplicity in specifying unit tests. JUnit is fully integrated in many *Integrated Development Environments* (IDE) such as Eclipse. It supplies a *Graphical User Interface* (GUI) which simplifies testing and gives valuable information about executed tests, occurred errors, and reported failures.

- **TTCN-3:** It is known as the only internationally standardized testing language by the *European Telecommunications Standards Institute* (ETSI). It is designed to satisfy many testing needs and to be applied to different types of testing. The strength of TTCN-3 relies on its platform independence. This makes the use of TTCN-3 more appropriate in the case of heterogeneous systems. In contrast to various testing and modeling languages, TTCN-3 does not comprise only a test language, but also a test system architecture for the test execution phase. In fact, this TTCN-3 test system comprises interacting entities that manage test execution, interpret or execute compiled TTCN-3 code and establish real communication with the SUT.

In this work, we adopt TTCN-3 as a convenient test notation and test execution support for validating dynamic and distributed systems.

2.3.4 Test architectures for distributed systems

A test architecture is composed of a set of *Testers* which are entities that interact with the SUT to execute the available test cases and to observe responses. A test case can be defined as a set of input values, execution preconditions, execution post-conditions and expected results developed generally in order to verify the conformance of a system to its requirements.

Test architectures can be either centralized (27; 28) or distributed (29; 30; 28). A centralized architecture consists of a single tester that communicates with the different ports of the system under test. In our work we consider a distributed test architecture by associating a new tester with each component of the SUT. More precisely we adopt the TTCN-3 standardized test architecture (26).

2.4 Testing dynamically adaptable systems

In the literature, two well-known testing techniques are usually performed to check the correctness of an evolved software system. *Regression tests* are executed after the occurrence of each modification at design time whereas *Runtime tests* are performed at service time.

2.4.1 Regression testing

Regression testing attempts to validate modified software and ensure that no errors are introduced into previously tested code (31). This technique guarantees that the modified program is still working according to its specification. It is commonly applied during the development phase and not at runtime. When the program code is modified code-based regression testing techniques can be advocated, as in (32).

According to Leung et al. (33), old tests can be classified into three kinds of tests:

- **Reusable tests**: valid tests that cover the unmodified parts of the SUT.
- **Retestable tests**: still valid tests that cover modified parts of the SUT.
- **Obsolete tests**: invalid tests that cover deleted parts of the SUT.

Leung et al. identify two types of regression testing. In the progressive regression testing, the SUT specification can be modified by reflecting some enhancements or some new requirements added in the SUT. In the corrective regression testing, only the SUT code is modified by altering some instructions in the program whereas the specification does not change. Thus, new tests can be classified into two classes:

- **New specification tests:** new tests generated from the modified parts of the specification.
- **New structural tests:** structural-based test cases that test altered program instructions.

Although regression testing techniques are not dedicated for dynamically adaptable systems, research done in this area is useful to obtain in a cost effective manner a relevant test suite validating behavioral changes.

2.4.2 Runtime testing

The runtime testing activity is defined in (34) as any testing method that is carried out on the final execution environment of a system when the system or a part of it is operational. It can be performed both at deployment-time and at service-time.

For systems whose architectures remain constant after their initial installation, there is obviously no need to retest the system when it has been placed in-service. On the contrary, if any change in the execution environment or the system behavior/architecture occurs, service-time testing becomes a necessity to verify and validate the new system in the new situation.

2.4.3 Runtime testability

According to IEEE std. 610.12 (35), testability is defined as the degree to which a system or a component facilitates the performance of tests to determine whether a requirement is met. Consequently, runtime testability is defined as an indicator of the effort needed to test the running software without affecting its functionalities or its environment. In this direction, some approaches, such as (36), focused on proposing mathematical methods for its assessment. Next we explain how runtime testability varies according to two main characteristics of the SUT: *Test Sensitivity* and *Test Isolation*.

2.4.3.1 Test sensitivity

It is a component property that indicates whether the component under test can be tested without unwanted side-effects. In particular, a component is called test sensitive when it includes some behaviors or operations that cannot be safely tested at runtime. In this case, the component is called untestable. In the opposite case it is called a testable component.

2.4.3.2 Test isolation

This solution is applied by test engineers in order to counter the test sensitivity problem and to prevent test processes from interfering with business processes. Many test isolation techniques are available to fulfill such aim:

- **Built-In Test approach:** The *Built-In Test* (BIT) paradigm consists in building testable components. To do so, components are equipped with a test interface which provides operations ensuring that the test data and business data are not mixed during the test process (37; 38).
- **Aspect-based approach:** This technique uses *Aspect Oriented Programming* (AOP). Unlike the BIT approach that embeds test cases into components, the aspect-based approach integrates such test scripts into a separate module, i.e., aspect. Thus, maintainability of the component and its capacity to check itself are improved.

- **Tagging components:** This technique consists in marking the test data with a special flag in order to discriminate it from business data (39). The component is then called *test aware*. The principal advantage of this method is that one component can receive production as well as testing data simultaneously.
- **Cloning components:** This mechanism consists in cloning the component under test before the start of the test activity. Thus, test processes are performed by the clone while business processes are performed by the original component. To clone components efficiently, we must also duplicate their dependencies, known as *Deep clone strategy* (40).
- **Blocking components:** In case of untestable components, a blocking strategy can be adopted as a test isolation technique. In fact, it consists in interrupting the activity of component sources for a lapse of time representing the duration of the test.

The listed test isolation techniques suffer from some weaknesses. First of all, the cloning strategy is very costly in terms of resources especially when the number of needed clones increases. Besides, BIT, tagging and aspect-based techniques have an additional development burden. Regarding the blocking option, it may affect the performance of the whole system, especially its responsiveness in case of real-time systems.

2.5 Related work on regression testing

In the literature, many researchers have investigated regression testing techniques to reestablish confidence in modified software systems. Their research spans a wide variety of topics, namely test selection, test prioritization, efficient test generation, etc. The existing approaches are classified into : code-based regression testing (41; 32; 42), model-based regression testing (43; 44; 45; 46; 47; 48) and software architecture-based regression testing (49; 50).

2.5.1 Code-based regression testing approaches

Rothermel et al. (32) construct control flow graphs from a program and its modified version and then use the elaborated graphs to select all non obsolete tests from the old test suite. The obtained set of tests is still valid and covers the changed code. Similarly, Granja et al. (41) deal with identifying program modifications and selecting attributes required for regression testing. Based on data flow analysis, the authors use the obtained elements to select retestable tests. Test generation features to produce new tests covering new behaviors are not discussed in this work. In (42), the authors apply a regression test selection and prioritization approach based

on code coverage. The obtained results show the efficiency of the proposed implementation to reveal defects, to reduce the test set and test time. Nevertheless, this approach is specific to C++ programming language. Moreover, it is tightly related to the system under test and cannot be easily applied to other systems.

2.5.2 Model-based regression testing approaches

Brian et al. (47) introduce a UML-based regression test selection strategy. The proposed approach automatically classifies tests issued from the initial behavioral models as obsolete, reusable and retestable tests. Identifying parts of the system that require additional tests to generate was not tackled by this approach. Similarly, the work of (48) deals with minimizing the impact of test case evolution by avoiding the regeneration of full test suites from UML diagrams. A point in favor of this work is the enhancement of the test classification already proposed by Leung et al. (33). In fact, the authors define a more precise test status based on the model dependence analysis. Pilskalns et al. (45) present a new technique that reduces the complexity of identifying the modification impact from various UML diagrams. This proposal is based on an integrated model called *Object Method Directed Acyclic Graph* built from class and sequence diagrams as well as *Object Constraint Language* (OCL) expressions. Chen et al. (46) propose a safe regression technique relying on *Extended Finite State Machine* (EFSM) as a behavioral model and a dependence analysis approach. Similar to (46), Korel et al. (51) support only elementary modifications, namely the addition and the deletion of a transition. In this context, they present two kinds of model-based test prioritization methods : selective test prioritization and model dependence-based test prioritization.

2.5.3 Software architecture-based regression testing

Harrold et al. (49) introduced the use of the formal architecture specification instead of the source code in order to reduce the cost of regression testing and analysis. This idea has been explored later by Muccini et al. (50). The authors propose an effective and well-implemented approach called *Software Architecture-based Regression Testing*. They apply regression testing at both code and architecture levels whenever the system implementation or its architecture evolve.

2.5.4 Discussion

Two major questions are identified when several regression testing approaches are studied. The first one is how to select a relevant and a minimal subset of tests from the original test suite.

The second one is how to generate new tests covering only new behaviors. Responding to these challenging questions requires both test selection and generation capabilities. In this respect, we notice that some approaches focus only on the test selection activity at the code level (32; 41) or at the model level (47) whereas the work of (46) deals only with model-based test generation issue. Addressing both activities as in (44; 48; 45; 50) is highly demanded in order to reduce their cost especially in terms of number of tests and time required for their execution.

Up to our knowledge, no previous work dealt with the use of regression testing approaches at runtime. Therefore, our goal was to handle test selection and test generation activities at a higher abstract level without code source access while the SUT is operational.

2.6 Related work on runtime testing

We identified several approaches supporting only runtime testing of structural adaptations (52; 53; 54; 55; 56; 57). The work presented in (58) deals only with behavioral adaptations. The approaches in (59; 60; 61) take into account both structural and behavioral adaptations while performing runtime tests. Next runtime testing approaches are discussed from various perspectives.

2.6.1 Supporting test isolation strategies

In the literature, several research approaches have a strong tendency to investigate test isolation concept in order to reduce the interference risk between test processes and business processes. The majority accommodates the Built-In Test paradigm for this purpose (34; 53; 39). Similarly to this strategy, the approaches introduced in (60) and (56) deal with putting all the involved components into a testing mode before the execution of runtime tests. We also identified approaches dealing with runtime testing of untestable components. They afford other test isolation strategies such as *Safe Validation with Adaptation* (58), which is equivalent to the blocking strategy already introduced. Similar to cloning components, the *Replication with Validation* strategy was proposed by (61) as a means of test isolation. Furthermore, instantiating services at runtime and using new service instances for runtime testing purposes is proposed by (57). Finally, the *Mobile Resource-Aware Built-In-Test* (MORABIT) framework introduced in (52) addresses the runtime testing of heterogeneous software systems composed of testable and untestable components. This framework supports two test isolation strategies: cloning if components under test are untestable and the BIT paradigm otherwise.

2.6.2 Handling test distribution

The test distribution over the network has been rarely addressed by runtime testing approaches. Most of the studied works assume that tests are integrated into components under test. We identified only two approaches that shed light on this issue.

First, the authors of (62; 59) introduce a light-weight framework for adaptive testing called *Multi Agent-based Service Testing* in which runtime tests are executed in a coordinated and distributed environment. This framework defines a coordination architecture that facilitates test agent deployment and distribution over the execution nodes and test case assignment to the adequate agents. Unfortunately, this framework suffers from a dearth of test isolation concerns.

In the second study (54), a distributed in vivo testing approach is introduced. This proposal defines the notion of *Perpetual Testing* which suggests the proceeding of software analysis and testing throughout the entire lifetime of an application. The main contribution of this work consists in distributing the test load in order to attenuate the workload and improve the SUT performance by decreasing the number of tests to run. However, this framework does not handle the occurrence of behavioral adaptations and supports only the cloning strategy for test isolation.

2.6.3 Handling test selection and evolution

The test selection issue has to be addressed seriously with the aim of reducing the amount of tests to rerun. One of the potential solutions that tackle this issue is introduced in (39). The proposed approach uses dependency analysis to find the affected components by the change.

In the literature, we distinguish the *ATLAS* framework (53), which affords a test case evolution through an *Acceptance Testing Interface*. This strategy ensures that tests are not built in components permanently and can evolve when the system under test evolves, too. The major limitation of this approach is the lack of automated test generation since tests are not generated automatically from components' models and specification. Contrary to this approach, the authors of (62) and (60) address this last issue. Both methods regenerate all test cases from new service specifications when dynamic behavioral adaptations occur. However, regenerating all tests can be costly and inefficient. To overcome these limitations, Akour et al. (63) propose a model-driven approach for updating regression tests after dynamic adaptations. Called *Test Information Propagation*, this proposal consists in synchronizing component models and test models at runtime. In the same context, Fredericks et al. (64) propose an approach that adapts test cases at runtime with the aim of ensuring that the SUT continues its execution safely and correctly when environmental conditions are adapted. Nevertheless, this work can only adapt test case parameters at runtime and it is not intended to dynamically add or remove test cases.

2.6.4 Affording platform independent test systems

The major test systems, that have been surveyed, have been implemented in a tightly coupled manner to the programming language of components or to the underlying component model such as Fractal (53), OSGi (57), the *Dynamic Adaptive System Infrastructure* (DAiSI) component model in (56) and the MORABIT component model in (65). Another approach presented in (54) affords a Java-based framework. Many other approaches (66; 67; 68; 69; 70; 71; 72; 73; 74) benefit from the strengths of the TTCN-3 standard as a platform independent language for specifying tests even for heterogeneous systems. However, they address testing issues at design time and not at runtime. Deussen et al. (75) stress the importance of using the TTCN-3 standard to build an online validation platform for internet services. However, this work neglects the test isolation issue.

2.6.5 Supporting test resource awareness

To the best of our knowledge, this problem has been studied only by Merdes'work (52). Aiming at adapting the testing behavior to the given resource situation, it provides a resource-aware infrastructure that keeps track of the current resource states. To do this, a set of resource monitors are implemented to observe the respective values for processor load, main memory, battery charge, network bandwidth, etc. According to resource availability, the proposed framework is able to balance in an intelligent manner between testing and the core functionalities of the components. It provides in a novel way a number of test strategies for resource aware test management. Among these strategies, we can mention, for example, *Threshold Strategy* under which tests are performed only if the amount of used resources does not exceed thresholds.

2.6.6 Discussion

Considering both structural and behavioral adaptations was only done by (59; 60; 61). Furthermore, we noticed a quasi-absence of approaches offering platform independent test systems except in (75). However, the authors of (75) support homogeneous systems-under test made up of only testable (or only untestable ones). We identified only one work (52) that deals with combining two test isolation strategies. Moreover, only the latter approach tackled the issue of resource limitations and time restriction during runtime testing. Regarding test evolution and generation at runtime, the authors of (62; 60) that regenerate all test cases from the new specifications when dynamic behavioral adaptations occur. The work of (64) tries to reduce this cost by adapting exiting test cases to the evolved environmental conditions but without generating new tests covering new behaviors or removing obsolete ones. To partially overcome

this limitation, (63) supports only reductive changes (e.g., removing existing components) and then adapts the test suite by removing obsolete tests and by updating the set of retestable tests.

In summary, our study on runtime testing approaches reveals a dearth in the provision of a platform-independent support for test generation and execution which considers resource limitations and time restriction. To surmount this major lack, our ultimate goal was to conceive a safe and efficient framework that minimizes the cost of checking a running system after each dynamic adaptation either structural or behavioral. From the test execution perspective, setting up a TTCN-3 test system for the distribution, isolation and execution of a minimal set of test cases identified after the occurrence of structural adaptations is strongly required. From the test generation perspective, proposing a selective test case generation method that derives test cases efficiently from the affected parts of the SUT behavioral model and selects relevant tests from the old test suite was a must, too.

2.7 Summary

This chapter addressed the fundamentals related to runtime validation of dynamically adaptable systems. It was mainly dedicated to give an overview of the most common concepts frequently used in the field of software testing, especially while testing evolvable systems. In this context, two well-known techniques, namely regression testing and runtime testing were introduced. Morover, we discussed the state of art of testing modified systems. Research done in the area of regression testing as well as runtime testing was analyzed.

CHAPTER 3

Runtime Testing for Structural Adaptations

3.1 Introduction

Testing at design-time or even at deployment-time usually demonstrates that the System Under Test, SUT, satisfies its functional and non-functional requirements. However, its applicability becomes limited and irrelevant when this system is adapted at runtime according to evolving requirements and environmental conditions that were not explicitly specified at design-time. For this reason, runtime testing is strongly required to extend assurance from design-time to runtime.

The rest of this chapter is structured as follows. In Section 3.2, a brief overview of the overall runtime testing process is given. First of all, the timing cost is reduced by executing only a minimal subset of test cases that validates the affected parts of the system by dynamic changes. In this respect, Sections 3.3 and 3.4 introduce the use of the dependency analysis technique to identify the affected parts of the dynamically adaptable system and their corresponding test cases. Secondly, Section 3.5 introduces the method we use to effectively distribute the obtained tests over the network with the aim of alleviating runtime testing load and not disturbing SUT performance. Thirdly, Section 3.6 presents the standard-based test execution platform that we designed for test isolation and execution purposes. The latter is extended to supply a test isolation layer that reduces the interference risk between test processes and business processes. Ultimately, this chapter is concluded in Section 3.7.

3.2 The Approach in a nutshell

The process depicted in Figure 11.1 spans the different steps to fulfill with the aim of executing runtime tests when structural reconfiguration actions are triggered, as follows :

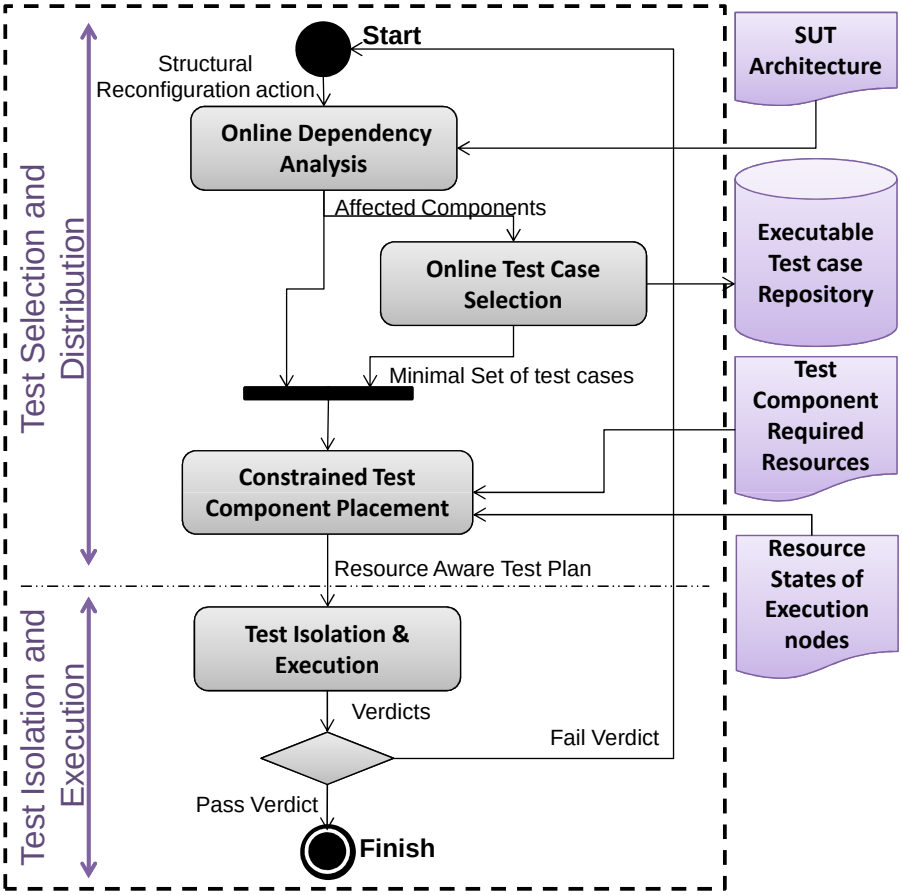


Figure 3.1: Runtime testing process for the validation of structural adaptations.

- **Online Dependency Analysis:** In this step, we focus on identifying the affected components and compositions by a structural reconfiguration action.
- **Online Test Case Selection:** Once the affected parts of the system are identified, we look for their corresponding test cases that are stored in the *Executable Test Case Repository*.
- **Constrained Test Component Placement:** Test components are assigned to execution nodes in an appropriate manner with respect to resource and connectivity constraints.
- **Test Isolation and Execution:** A test isolation layer is set up then test components are dynamically created and test cases are executed.

More details are presented in the next sections.

3.3 Online dependency analysis

To reduce the time cost and the resource burden of the runtime testing process, the key idea is to avoid the re-execution of all tests at runtime when structural adaptations occur. Thus, we use the dependency analysis approach with the aim of determining the parts of the system impacted by dynamic evolutions and then computing a minimal set of tests to rerun. In fact, the dependency analysis technique is widely used in various software engineering activities including testing (76), maintenance and evolution (77; 78).

3.3.1 Definition

Dependencies between components is defined in (77) as "the reliance of a component on other(s) to support a specific functionality". It is also considered as a binary relation between two components. A component A is an antecedent to another component B if its data or functionalities are utilized by B. Equivalently, A component B is a dependent on another component A if it utilizes data or functionalities of A. Formally, the relation $\rightarrow$ called "*Depends on*" is defined in (79) where $B \rightarrow A$ means that the component B depends on the component A. The set of all dependencies in a component-based system is defined as : $\mathcal{D} = \{(C_i, C_j) : C_i, C_j \in \mathcal{S} \wedge C_i \rightarrow C_j\}$ where $\mathcal{S}$ is the set of components in the system. Accordingly, the current system configuration is a set of components and its dependencies $Con = (\mathcal{S}, \mathcal{D})$.

Several forms of dependencies component-based systems are identified in the literature (76). For instance, we mention data dependency (i.e., data defined in one component is used in another component), control dependency (i.e., caused by sending a message from one component to another component), etc. The main dependency form that we support in this work is the interface dependency, which means that a component requires (respectively provides) a service from (respectively to) another component.

3.3.2 Dependency representation

To represent and analyze component dependencies, two formalisms are generally described : a *Component Dependency Graph* (CDG) and a *Component Dependency Matrix* (CDM). A CDG is a directed graph denoted by $\mathcal{G} = (\mathcal{S}, \mathcal{D})$ where: $\mathcal{S}$ is a finite nonempty set of vertices representing system's components and $\mathcal{D}$ is a set of edges between two vertices, $\mathcal{D} \subseteq (\mathcal{S} \times \mathcal{S})$. A CDM is defined as a 0-1 Adjacency Matrix $\mathcal{AM}_{n \times n}$, that represents direct dependencies in a component-based system. In this matrix, each component is represented by a column and a row. If a component C_i depends on a component C_j then $d_{ij} = 1$ otherwise $d_{ij} = 0$. Figure 3.2 shows an example of dependency graph and its corresponding adjacency matrix.

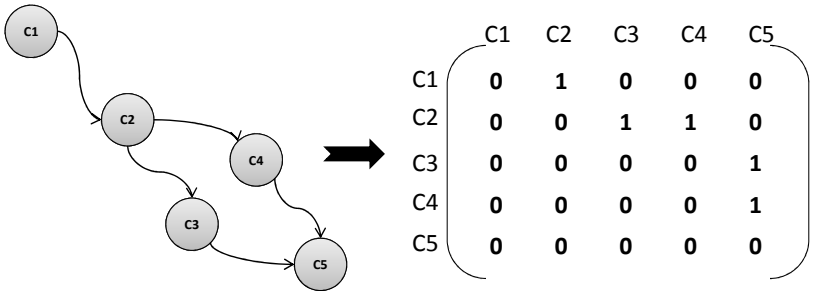


Figure 3.2: A CDG and its CDM representing direct dependencies.

Initially, $\mathcal{D}$ represents only direct dependencies between components. In order to gather all indirect dependencies in the component-based system, the transitive closure of the graph has to be calculated. Several transitive closure algorithms have been widely studied in the literature such as the Roy-Warshall algorithm and its modification proposed by Warren (80).

3.4 Online test case selection

This concern has been extensively studied in the literature. In fact, various regression test selection techniques have been proposed with the purpose of identifying a subset of valid test cases from an initial test suite that tests the affected parts of a program. These techniques usually select regression tests based on data and control dependency analysis (81).

Two kinds of tests are considered after the occurrence of dynamic adaptations. On the one hand, unit tests are executed to validate individual affected components. On the other hand, integration tests are performed to check interactions and interoperability between components.

Let us take an example with four components and a dependency graph that looks like Figure 3.3. Assume that C_2 is replaced with a new version. Thus, two dependence paths are identified: $C_1 \rightarrow C_2 \rightarrow C_3$ and $C_1 \rightarrow C_2 \rightarrow C_4$. As a result, the mapping to integration tests produces: $ITC1C2C3$ and $ITC1C2C4$ have to be rerun.

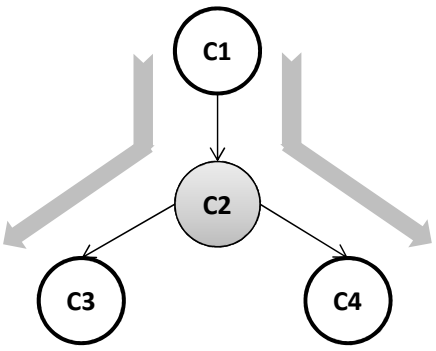
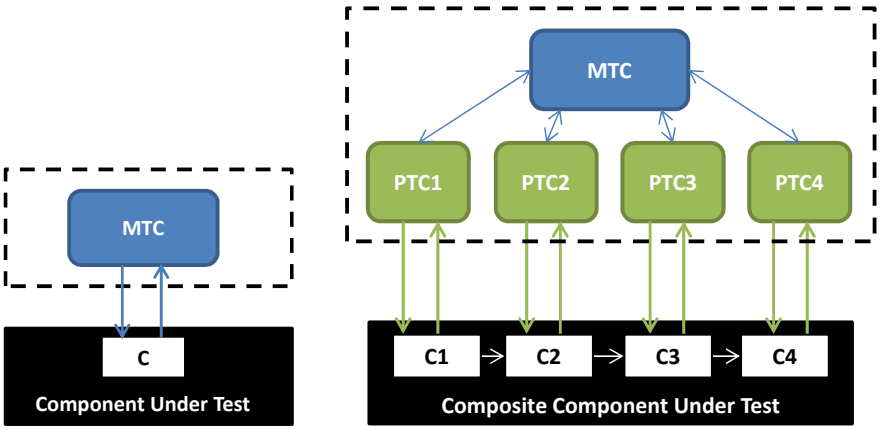


Figure 3.3: Illustrative example of dependence path computation.

Recall that tests are written in the TTCN-3 notation and are executed by TTCN-3 test components. As depicted in Figure 3.4a, an MTC component is only charged with executing a unit test. It shares this responsibility with other PTC components when an integration test is executed (see Figure 3.4b). Each PTC is created to simulate a test call from a component to another at lower hierarchy in the dependence path. The following subsection copes with test case distribution and more precisely with main test components assignment to execution nodes.



(a) Unit test configuration. (b) Integration test configuration.

Figure 3.4: TTCN-3 test configuration for unit and integration testing.

3.5 Constrained test component placement

In the following subsections, we discuss how to formalize resource and connectivity constraints and how to find the adequate deployment host for each test involved in the runtime testing process.

3.5.1 Resource allocation issue

For each node in the execution environment, three resources are monitored during the SUT execution: the available memory, the current CPU load and the battery level. The value of each resource can be directly captured on each node through the use of internal monitors. These values are measured after the runtime reconfiguration and before starting the testing activity. Formally, provided resources of m execution nodes are represented through three vectors: C contains the CPU load, R provides the available RAM and B introduces the battery level.

$$C = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix} \quad R = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{pmatrix} \quad B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

The resources required by the n test components are initially computed at the deployment time after a preliminary test run. Similarly, they are formalized over three vectors : D_c that contains the required CPU, D_r that introduces the required RAM and D_b that contains the required battery by each test.

$$D_c = \begin{pmatrix} dc_1 \\ dc_2 \\ \vdots \\ dc_n \end{pmatrix} \quad D_r = \begin{pmatrix} dr_1 \\ dr_2 \\ \vdots \\ dr_n \end{pmatrix} \quad D_b = \begin{pmatrix} db_1 \\ db_2 \\ \vdots \\ db_n \end{pmatrix}$$

As the proposed framework is resource aware, The overall resources required by n test components must not exceed the available resources in m nodes. This rule is formalized as follows:

$$\begin{cases} \sum_{i=1}^{n} x_{ij}\, dc_i \leq c_j & \forall\, j \in \{1, \cdots, m\} \\ \sum_{i=1}^{n} x_{ij}\, dr_i \leq r_j & \forall\, j \in \{1, \cdots, m\} \\ \sum_{i=1}^{n} x_{ij}\, db_i \leq b_j & \forall\, j \in \{1, \cdots, m\} \end{cases} \tag{3.1}$$

where the two dimensional variable x_{ij} can be equal to 1 if the corresponding test component i is assigned to the node j, 0 otherwise.

3.5.2 Connectivity issue

Dynamic environments are characterized by frequent and unpredictable changes in connectivity caused by firewalls, non-routing networks, node mobility, etc. For this reason, we have to pay attention when assigning a test component to a host computer by finding at least one route in the network to communicate with the component under test. For each test component, a set of forbidden nodes to discard during the constrained test component placement step is defined. This connectivity constraint is denoted as follows:

$$x_{ij} = 0 \quad \forall\, j \in \mathit{forbiddenNodeSet}(i) \tag{3.2}$$

Finding a satisfying test placement solution is achieved by fitting the former constraints (3.1) and (3.2). The latter can be seen as a *Constraint Satisfaction Problem* (CSP) (82).

3.5.3 Optimizing the test component placement problem

Looking for an optimal test placement solution consists in identifying the best node to host the concerned test component in response with two criteria : its distance from the node under test and its link bandwidth capacity. To do so, we are asked to attribute a profit value p_{ij} for assigning the test component i to a node j. For this aim, a matrix $\mathcal{P}_{n\times m}$ is computed as follows:

$$p_{ij} = \begin{cases} 0 & \text{if} \quad j \in \mathit{forbiddenNodeSet}(i) \\ \mathit{maxP} - k \times \mathit{step}_p & \text{otherwise} \end{cases} \tag{3.3}$$

where maxP is a constant, $\mathit{step}_p = \frac{\mathit{maxP}}{m}$, k corresponds to the index of a node j in a *Rank Vector* that is computed for each node under test. This vector corresponds to a classification of the connected nodes according to two criteria : the distance from the testing node to the node under test (83) and the link bandwidth capacities.

As a result, the constrained test component placement module generates the best deployment host for each test component involved in the runtime testing process by maximizing the total profit value while fitting the former resource and connectivity constraints. Thus, this problem is formalized as a variant of the Knapsack Problem, called *Multiple Multidimensional Knapsack Problem* (MMKP).

$$MMKP = \begin{cases} \mathit{maximize} \quad \mathcal{Z} = \sum_{i=1}^{n}\sum_{j=1}^{m} p_{ij}x_{ij} & (3.4) \\ \mathit{subject} \quad \mathit{to} \quad (3.1) \quad \mathit{and} \quad (3.2) & \\ \sum_{j=1}^{m} x_{ij} = 1 \quad \forall\, i \in \{1,\cdots,n\} & (3.5) \\ x_{ij} \in \{0,1\} \quad \forall\, i \in \{1,\cdots,n\} \quad \mathit{and} \quad \forall\, j \in \{1,\cdots,m\} & \end{cases}$$

Constraint (3.4) corresponds to the objective function that maximizes test component profits while satisfying resource (3.1) and connectivity (3.2) constraints. Constraint (3.5) indicates that each test component has to be assigned to at most one node.

3.6 Test isolation and execution support

With the purpose of alleviating the complexity of testing adaptable and distributed systems, we propose a test system called, ***TT****CN-3 test system for* ***R****untime* ***T****esting* (TT4RT) (84).

3.6.1 Detailed interactions of TT4RT components

TT4RT relies on the classical TTCN-3 test system. Thus, it reuses all its constituents, namely Test Management (TM), TTCN-3 Executable (TE), Component Handling (CH), Coding and Decoding (CD), System Adapter (SA) and Platform Adapter(PA). These entities are briefly introduced below. As depicted in Figure 3.5, a new *Generic Test Isolation Component* is added to the TTCN-3 reference architecture with the aim of handling test isolation concerns. The next steps define the different components of TT4RT and their internal interactions:

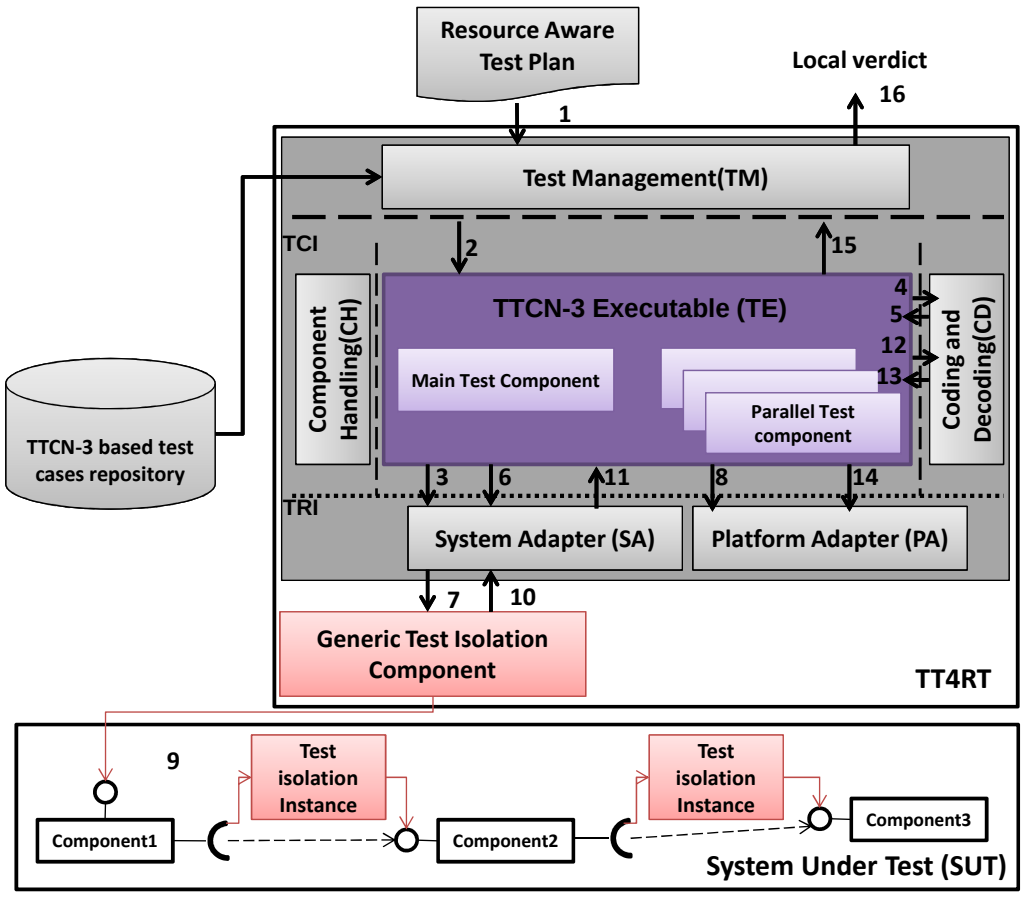


Figure 3.5: Internal interactions in the TT4RT system.

When a reconfiguration action is triggered, the RATP (Ressource Aware Test Plan) file is generated and it is considered as an input to the TT4RT test system (Step **1**). The test execution is initiated by the TM entity which is charged with starting and stopping runtime tests (Step **2**). Once the test process is started, the TE entity (i.e., which is responsible of executing the compiled TTCN-3 code) creates the involved test components and informs the SA entity (i.e., which is charged with propagating test requests from TE to SUT) with this start up in order to set up its communication facilities (Step **3**). Next, TE invokes the CD entity in order to encode the test data from a structured TTCN-3 value into a form that will be accepted by the

SUT (Step **4**). The encoded test data is passed back to the TE entity as a binary string and forwarded to the SUT via the SA entity (Steps **5-6-7**). After the test data is sent, a timer can be started (Step **8**). The Generic Test isolation Component, implementing test isolation facilities, intercepts the test request, identifies the component under test and its supported test isolation technique and prepares the test environment (Steps **7-9**). Different test isolation instances are automatically created to perform test isolation inter-component invocations (Step **9**). The SUT response is forwarded to the SA entity through the Generic Test Isolation Component. The given response is an encoded value that has to be decoded in order to be understandable by the TTCN-3 test system (Step **10**). For this purpose, the SA entity forwards the encoded test data to the TE entity (Step **11**). The TE entity transmits the encoded response to the CD entity with the intention of decoding it into a structured TTCN-3 value (Step **12**). The decoded response is passed back to the TE that stops the running timer and finally computes a verdict (pass, fail or inconclusive) for the current test case (Steps **13-14-15**). Finally, a local verdict is computed depending on the obtained verdicts for test cases executed by the current TT4RT instance (Step **16**).

3.6.2 Overview of the Generic Test Isolation Component

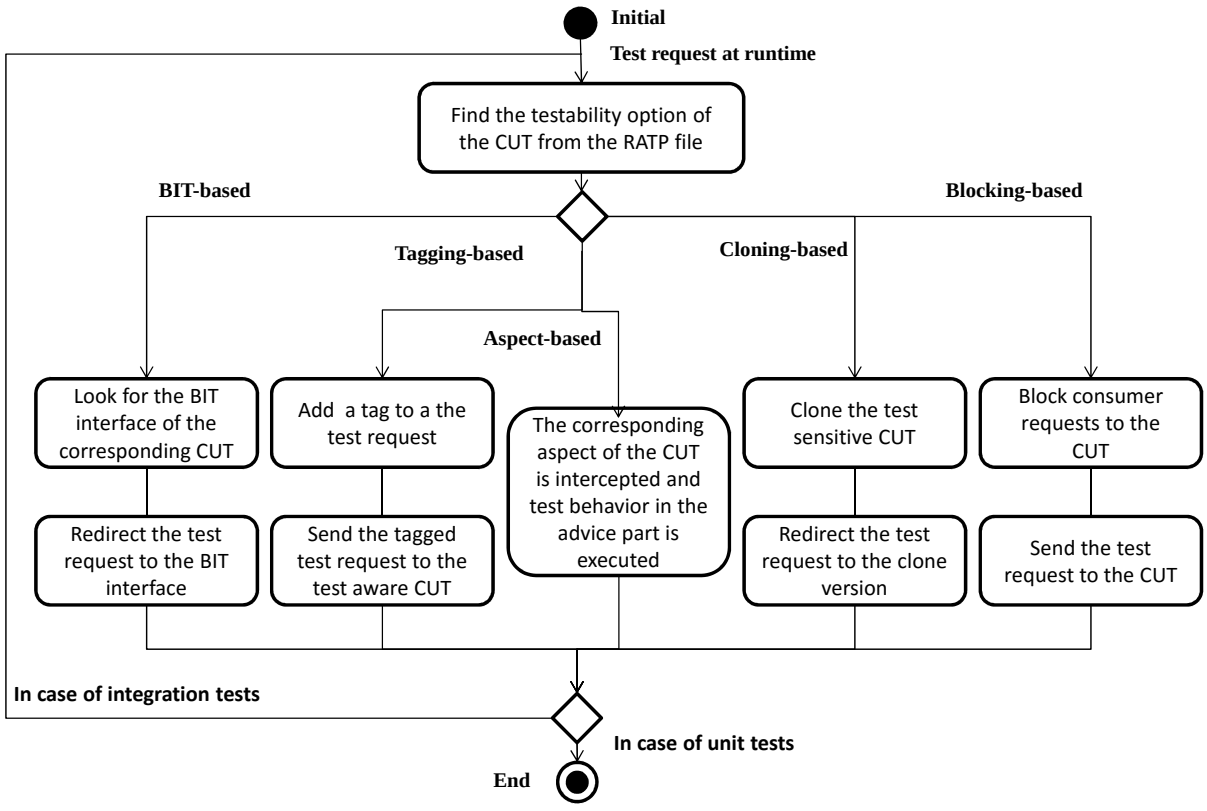


Figure 3.6: Test isolation policy.

As outlined in Figure 3.6, the proposed policy is executed while a test request is intercepted from the System Adapter entity. Five strategies can be applied in response to the testability degree of a Component Under Test (CUT). With the assumption that the CUT is testable, the test request can be redirected to one or more test operations provided by its corresponding test interface or its associated aspect (particularly in the advice part) when the aspect-based technique is used. If the component under test is test aware, the tagging technique is applied and the CUT is invoked by tagging the input test data with a flag to discriminate them from business data. If we deal with untestable components, either cloning or blocking techniques can be performed. For a test sensitive component, a clone is created and the test request is redirected to it. Regarding the blocking strategy, it consists in interrupting the activity of the component under test consumers for a lapse of time that corresponds to the test duration. During this period, all business requests are delayed until the end of the test. Once the test is achieved, the component under test consumers are unlocked and the delayed requests are treated.

3.6.3 The adopted distributed architecture

The TTCN-3 standard offers concepts related to test configurations, test components, their communicating ports between each other and with the SUT, their execution and their termination only at an abstract level. Nevertheless, the means to control the distributed execution of these test components are not explicitly defined in the current specification. Regarding this issue, we propose our own test architecture that relies on a *Test System Coordinator* (TSC) and several TT4RT instances. As outlined in Figure 3.7, TSC is mainly charged with distributing selected test cases to rerun and assigning their corresponding test components to the execution nodes. Several TT4RT instances are installed within the host computers involved in the final execution environment. They can be seen as test containers that hold test components (i.e., either MTC or PTC components). Each instance controls the execution of a subset of selected test cases.

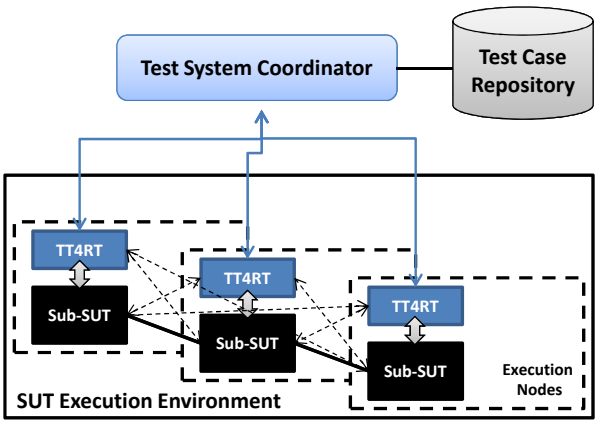


Figure 3.7: The distributed test execution platform.

3.7 Summary

In this chapter, we applied the runtime testing process to validate component-based systems after the occurrence of dynamic structural adaptations. For this aim, we proposed a generic and resource aware test execution platform that covers essentially two phases. The first phase deals with test selection and distribution concerns. The main issue tackled in this first part is alleviating test burden, cost and resource consumption. This goal is achieved by reducing the amount of test cases to rerun and by assigning efficiently their associated test components to execution nodes while fitting resource and connectivity constraints. The second phase handles test isolation and execution concerns. Based on the TTCN-3 standard, we proposed a test system, TT4RT, which performs tests written in a standardized notation. Accordingly, we gained in terms of using the same notation for all types of tests and using a generic and flexible test harness. Furthermore, TT4RT afforded a test isolation infrastructure supporting components with various testability options (i.e., testable, test aware, untestable, etc.).

CHAPTER 4

Runtime Testing of Behavioral Adaptations

4.1 Introduction

Running old test suites on dynamic software systems, in which not only the structure evolves but also the behavior may change, seems to be meaningless. Therefore, it is highly required to update test suites in a cost effective manner as long as the software system is changing to fulfill new requirements. In this chapter, we address this issue by merging model-based testing and selective regression testing capabilities. To do so, we propose a Selective Test Generation Approach, called TestGenApp. The latter is briefly outlined in Section 4.2. Background materials on timed automata, UPPAAL formalism and observer automata are presented in Section 4.3. Then, we introduce the model differencing algorithm in Section 4.4. The output of this module is then used in Section 4.5 to perform an old test classification. Section 4.6 handles the test coverage customization that we propose in order to generate efficiently new tests. Moreover, it presents the algorithm that we propose to adapt either aborted or obsolete tests. Once abstract test sequences are obtained, their mapping to the TTCN-3 notation is discussed in Section 4.7. Finally, this chapter is concluded in Section 4.8.

4.2 The approach in a nutshell

As illustrated in Figure 4.1, our Selective Test Generation Approach is composed of four modules.

- **Model Differencing Module:** It is proposed to capture correspondences and differences

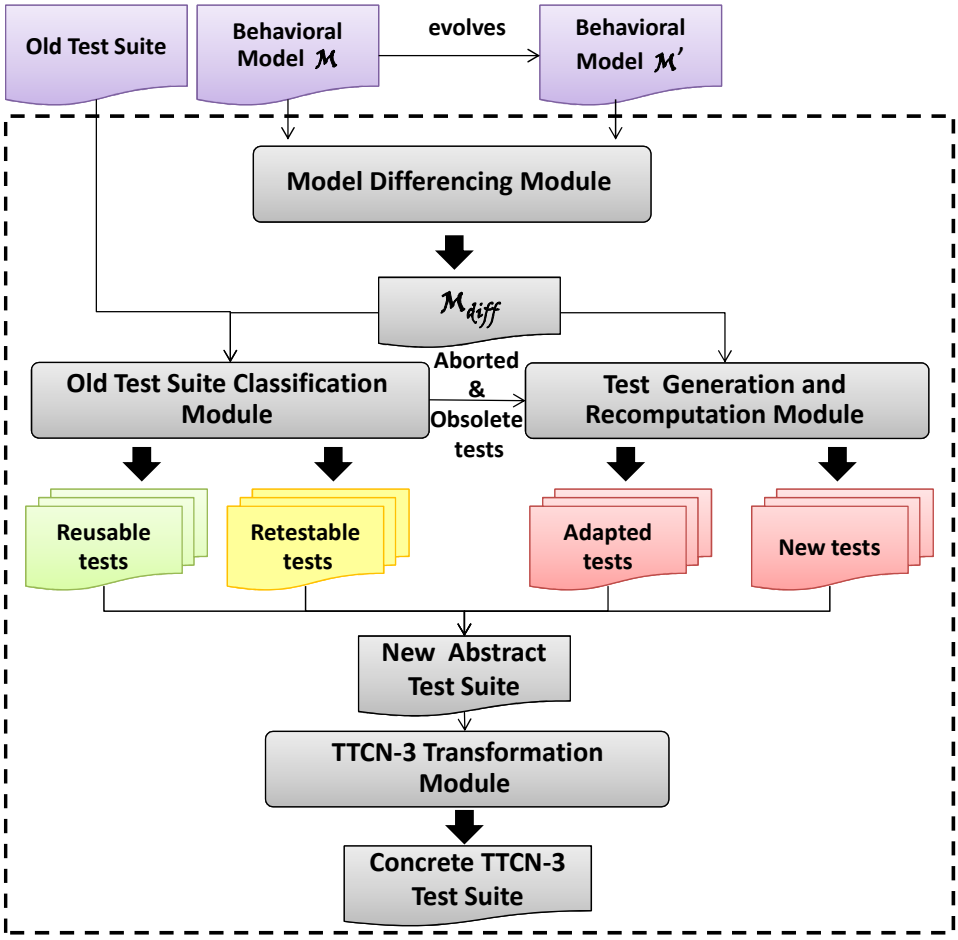


Figure 4.1: TestGenApp: Selective test case generation approach.

between two models in terms of added, removed or modified locations and transitions.

- **Old Test Suite Classification Module:** It is charged with classifying the old test suite issued from the original model $\mathcal{M}$ into *reusable*, *retestable*, *aborted* and *obsolete* tests.
- **Test Generation and Recomputation Module:** It generates new abstract test sequences covering newly added behaviors and adapts aborted and obsolete tests.
- **TTCN-3 Transformation Module:** It is used to transform the abstract test sequences, obtained in the last step, into the TTCN-3 code.

4.3 Prerequisites: UPPAAL Timed Automata

In order to specify the behavioral models of evolved systems, *Timed Automata* (TA) is chosen for the reason that it is a widespread formalism usually used for modeling behaviors of critical and real-time systems. More precisely, we opt for the particular UPPAAL style (85) 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 $\mathcal{C}$ be a set of variables called clocks, and $\mathcal{A}\text{ct} = \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$ with $\mathcal{I}$ a set of input actions, $\mathcal{O}$ a set of output actions , and the non-synchronizing action (denoted τ). Let $\mathcal{G}(\mathcal{C})$ denote the set of guards on clocks being conjunctions of constraints of the form $c \bowtie n$, where $c \in \mathcal{C}$, $n \in \mathsf{N}$, and $\bowtie \in \{\leqslant, \leq, =, \geq, \geqslant\}$. Moreover, let $\mathcal{U}(\mathcal{C})$ denotes the set of updates of clocks corresponding to sequences of statements of the form $c := n$.

A timed automaton over $(\mathcal{A}\text{ct}, \mathcal{C})$ is a tuple $(L, l_0, \mathcal{A}\text{ct}, \mathcal{C}, I, E)$, where :

- L is a set of locations, $l_0 \in L$ is an initial location.
- $I : L \longmapsto \mathcal{G}(\mathcal{C})$ a function that assigns to each location an invariant.
- E is a set of edges such that $E \subseteq L \times \mathcal{G}(\mathcal{C}) \times \mathcal{A}\text{ct}_\tau \times \mathcal{U}(\mathcal{C}) \times L$

We write $l \xrightarrow{g,\alpha,u} l'$ when $\langle l, g, \alpha, u, l' \rangle \in E$.

Let $(L, l_0, \mathcal{A}\text{ct}, \mathcal{C}, I, E)$ be a timed automaton. The semantics of TA is defined in terms of a timed transition system over states in the form (l, σ) where l is a location and $\sigma \in \mathbb{R}^{\mathcal{C}}_{\geqslant 0}$ is a clock valuation satisfying the invariant of l. The initial state (l_0, σ_0) is a state where l_0 is the initial location of the automaton and σ_0 is the initial mapping where $\forall\, c \in \mathcal{C}, c = 0$. Indeed, there are two kinds of transitions :

- Delay transitions, $(l, \sigma) \xrightarrow{d} (l, \sigma + d)$, in which all clock values of the automaton are incremented with the amount of the delay, denoted $\sigma + d$. In such a case, the automaton may stay in a location l as long as its invariant remains true.
- Discrete transitions, $(l, \sigma) \xrightarrow{\alpha} (l', \sigma')$, correspond to the execution of edges (l, g, α, u, l') for which the guard g is satisfied by σ. The clock valuation σ' of the target state is obtained by modifying σ according to updates u.

A run of timed automaton $(L, l_0, \mathcal{A}\text{ct}, \mathcal{C}, I, E)$ is a sequence of transitions $(l_0, \sigma_0) \xrightarrow{d_1} \xrightarrow{\alpha_1} (l_1, \sigma_1) \xrightarrow{d_2} \xrightarrow{\alpha_2} \ldots \xrightarrow{d_n} \xrightarrow{\alpha_n} (l_n, \sigma_n)$, with $\sigma_i \in \mathbb{R}^{\mathcal{C}}_{\geqslant 0}$, $d_i \in \mathbb{R}_{\geqslant 0}$ and $\alpha_i \in \mathcal{A}\text{ct}$. A network of timed automata, $TA_1 \| \ldots \| TA_n$ over $(\mathcal{A}\text{ct}, \mathcal{C})$ is modeled as a timed transition system obtained by the parallel composition of n TA over $(\mathcal{A}\text{ct}, \mathcal{C})$. Synchronous communication between the timed automata is performed by hand-shake synchronization using input and output actions.

4.4 Differencing between behavioral models

We introduce a novel *Differencing Algorithm* that concisely captures differences and similarities between networks of timed automata. In such a case, two main elements are compared: locations and transitions. First, we differentiate automata at the transition level. The two transitions T_i in the initial $\mathcal{TA}$ and T_j in the evolved $\mathcal{TA}'$ are considered similar if the following conditions are met :

a. T_i and T_j have the same source and target locations, and

b. they have the same values in the guard, assignment and synchronization fields.

The procedure used for this purspose takes as input two array lists including transitions of two timed automata : $\mathcal{TA}$ and $\mathcal{TA}'$. For each transition in the initial automaton, we firstly check its presence within the evolved one. From a technical point of view, this condition is checked by looking for an equivalent transition in the evolved model having similar source location *id* and target location *id*. As long as this condition is satisfied, we look for meeting conditions defined above meaning that they have the same source and target locations (i.e., name, label, committed, and urgent) and unchanged transition labels (i.e., guard, assignment and synchronization). As a result, the transition is considered unmodified.

If at least one condition is not respected, the transition is considered modified and it is marked in Yellow. New transitions which exist only in the evolved model are finally marked in Red. If a transition in $\mathcal{TA}$ does not have an equivalent in the new timed automaton $\mathcal{TA}'$, then this transition is not copied in the final array list because it is considered as a removed transition. The output of this procedure is an array list containing all marked transitions (unmodified, modified and new ones).

Following the same logic, we compare locations in both models. Two locations l_i in $\mathcal{TA}$ and l_j in $\mathcal{TA}'$ are considered similar if the following conditions are satisfied :

a. l_i and l_j have the same name and the same identifier,

b. they have the same incoming and outgoing transitions, and

c. they have the same invariant expression.

One location is marked as changed if at least one of these conditions is not met. Finally since the SUT is generally modeled by a network of timed automata, it is necessary to apply these procedures for each timed automaton in the network.

4.5 Old test suite classification

Inspired from the test classification proposed by Leung et al. (33), we introduce in this section a new test classification algorithm in which the old test suite generated from the original model $\mathcal{M}$ is analyzed and then partitioned into :

- Reusable test set T_{Ru} : valid traces that traverse unimpacted items by the change.
- Retestable test set T_{Rt} : valid traces that traverse impacted items by the change.
- Aborted test set T_{Ab} : invalid traces that cannot be animated on the new model because they cannot traverse modified items.
- Obsolete test set T_{Ob} : invalid traces that cannot be animated on the new model because they traverse removed items.

For that aim, each trace in the $\mathcal{TR}$ set should be animated on the $\mathcal{M}_{diff}$ model and its covered items should be identified. Two scenarios are then tackled. On the one hand, the test animation on the new model is achieved successfully. If the *trace* traverses unchanged items, it is classified as a *reusable test*. Otherwise, it is classified as a *retestable test*. On the other hand, the test animation on the new model is abandoned. If this abort is due to some removed items which are no longer available in the new model, the *trace* is seen as an obsolete test and it should be automatically discarded from the new test suite. Otherwise, this abort can be due to a modified transition which cannot be reached any more. In such a case, the *trace* is classified as an *aborted test*.

4.6 Test generation and recomputation

Our approach identifies critical regions in the evolved model not only by marking added locations and transitions but also by detecting old traces that cannot be animated on the new model. Consequently, the $\mathcal{M}_{diff}$ is used in this stage to generate new tests and adapt aborted and obsolete ones in a cost effective manner.

4.6.1 Test generation

To generate new tests covering newly added behaviors, we are based on the findings of Blom et al. (86), which express coverage criteria by using observer automata with parameters and formulate the test generation problem as a search exploration problem. Instead of adding auxiliary variables to enable the expression of a coverage criterion as a reachability property using

UPPAAL, the superposition of an observer onto timed automata is supported. The test generation tool UPPAAL CO$\sqrt{}$ER (87) supports the concept of observers and the test case generation algorithm (88). This efficient test suite generator is adopted in this thesis to realize a selective test generation approach when behavioral adaptations occur. The key idea is to formulate an observer that monitors only new regions in the evolved model. A test sequence satisfies this coverage criterion if when executed on the model it traverses at least one new edge where the *col* variable is updated to zero.

4.6.2 Test recomputation

At this stage, the new test suite $\mathcal{NTS}$ contains reusable, retestable and new tests.

$$\mathcal{NTS} = T_{Ru} \cup T_{Rt} \cup T_{New}$$

As mentioned before, the test animation on the evolved model may not be achieved due to some removed or modified items (i.e., locations or transitions) that cannot be traversed anymore. The key idea here consists in starting the test recomputation not from the initial state of the evolved model but from the last reachable state detected during the test animation. To do so, we take as inputs the current test suite $\mathcal{NTS}$, the evolved model $\mathcal{M}_{diff}$, the valid sub-trace $\mathcal{TR}$ from a given aborted trace (respectively obsolete trace) and the last reached state. Next, we look for the adjacency matrix of each timed automaton in $\mathcal{M}_{diff}$. Then, we explore the state space while generating all sub-paths that start from the given state and reach the initial one . For each sub-path, an adapted trace is obtained and added to the $\mathcal{NTS}$ test suite while verifying that the test redundancy is avoided.

The greatest added value of this technique is not only the decrease of the test generation cost but also its ability to create a test suite based on the kind of change (i.e., made up of reusable, retestable, new and adapted tests).

$$\mathcal{NTS} = T_{Ru} \cup T_{Rt} \cup T_{New} \cup T_{Ad}$$

If the obtained test suite is still large, a test prioritization strategy can be adopted. In that case, a high priority should be attributed to tests that cover critical zones of the evolved model such as new and adapted tests.

4.7 Test case concretization

Before introducing our proposed transformation rules that we use to derive TTCN-3 test cases from the abstract test sequences which have been newly generated from UPPAAL CO$\sqrt{}$ER, we give a brief overview of exiting research dealing with this issue.

4.7.1 Related work on transforming abstract tests to TTCN-3 notation

In the last decade, several researchers have paid more attention to automatic test case generation, more particularly to the concretization and the execution of abstract test suites (89; 90; 91; 92; 93; 94). We can mention, for instance, the approach in (93) which describes the generation of TTCN-3 test suites specifically for the *Session Initiation Protocol* without using formal specifications. The obtained test case generator is included in a commercial tool developed by Ericsson.

Deriving executable tests from UML 2.0 models was proposed by (92). Based on a commercial tool usually used for interoperability testing of healthcare applications, this work generates TTCN-3 test behaviors from UML sequence diagrams whereas TTCN-3 test data are generated from two eHealth standards, namely *Health Level 7* which is generally used for data representation and *Integrating Healthcare Entreprise* which is used for describing interactions between medical devices. Similarly, the approach in (90) shows the translation of *Message Sequence Charts* elements to the TTCN-3 notation.

Following the same principles of model-driven engineering, (95) proposes an approach that deals with the model transformation of *UML 2.0 Test Profile* (U2TP) elements into an executable test code. Within this work, U2TP is adopted as a modeling language for the test case specification. Then, the models are transformed to the TTCN-3 language.

To our best knowledge, only the works in (91; 94) handle the derivation of TTCN-3 test cases from abstract test sequences which are generated from finite state machines. In this context, authors in (91) make use of another variant of UPPAAL called UPPAAL CORA. Similar to our approach, they obtain witness traces from extended finite state machines and perform their derivation to TTCN-3 notation. Also, the approach presented in (94) is close to our proposal as it deals with a variant of timed automata called *Labeled-Ports Timed Input/Output Automata*. The latter formalism is used to model the different port behaviors in a given multi-port system. Then, a test generation algorithm is proposed and the obtained test cases are transformed into TTCN-3 language.

Since there are no available tools which are able to realize automatically the mapping of test sequences, generated from formal specifications based on timed automata, to the TTCN-3 notation, we had to develop our own transformation rules as outlined in the following subsection.

4.7.2 Transformation rules from abstract test sequences to TTCN-3

At this stage, we define several rules to derive TTCN-3 test cases from abstract test sequences (see Table 4.1) (96). First of all, we assume that for each test suite, a TTCN-3 module should be generated (**R1**).

Table 4.1: TTCN-3 transformation rules.

Rules	**Abstract concepts**	**TTCN-3 concepts**
R1	a test suite	a TTCN-3 module
R2	a single trace	a TTCN-3 test case
R3	Time dependent behavior	a timer definition
R4	a test sequence in the form of input delay output	a TTCN-3 test behavior
R5	each involved TA	a PTC component
R6	each channel	a template

Within the TTCN-3 standard, the module concept is used as a top-level structure. The first part includes definitions of test data, templates, test components, functions, communication ports, test cases and so on. The second part is usually used to describe the execution sequence of test cases. A test component can be either a Main Test Component (MTC) or a Parallel Test Component (PTC). Remember that the MTC is charged with creating PTC components and executing TTCN-3 test cases. To do so, a port must be defined in order to specify a *Point of Control and Observation* via which the test component can interact with other components and with the SUT. To specify time delays, TTCN-3 supports a timer mechanism (**R3**). Timers can be declared in component type definitions, the module control part, test cases, functions and altsteps. The channels declared in the UPPAAL XML file are transformed into TTCN-3 templates (**R6**). Moreover, an abstract test system interface is defined similarly to a component definition. It includes a list of all possible communication ports through which the test system is connected to the SUT. Once the test configuration is generated, we look for the mapping of the abstract test sequences to test cases. As stated in Table 4.1, for each test behavior in the form of *input delay output* a TTCN-3 function is derived (**R4**). Moreover, for a single trace (i.e., an abstract test sequence), a test case is generated (**R2**). Then, the communication is established between the PTC ports and the System ports. Finally, a sequence of calls to the already generated TTCN-3 functions is performed. To compile the obtained test cases, the TThree compiler (97) is used. It transforms the *Abstract Test Suite* into an *Executable Test Suite*. Then, our TT4RT test system can be used for test isolation and execution purposes.

4.8 Summary

The contributions presented in this chapter are many-fold. First, we defined a model differencing technique that highlights similarities and differences between an original behavioral model and the evolved one, generally obtained after behavioral adaptations. Second, we provided a test classification technique that selects efficiently reusable and retestable tests, identifies aborted tests and discards obsolete ones. These two steps are responsible for identifying critical regions in the evolved model that need to be covered by newly generated tests. For this purpose, we specified our own coverage criteria based on the observer automata language and we used the well-established tool UPPAAL model-checker and its extension UPPAAL CO$\sqrt{}$ER for generating new tests. Also, a test recomputation technique was introduced with the aim of adapting aborted and obsolete tests. Finally, the mapping of the abstract test sequences to the TTCN-3 notation was handled.

CHAPTER 5

Prototype Implementation

5.1 Introduction

This chapter deals with the implementation details of the proposed approach either when structural adaptations or behavioral adaptations take place. To this end, we provide a *Runtime Testing Framework for Adaptable and Distributed Systems* (RTF4ADS) that gathers the different modules already introduced in the previous chapters.

The rest of this chapter is structured as follows. Section 5.2 summarizes from a technical point of view the different constituents of RTF4ADS. Next, each implemented graphical user interface is illustrated in Sections 5.3 (Test selection and distribution GUI), 5.4 (Test isolation and execution GUI) and 5.5 (Selective Test Generation GUI). In Section 5.6 reports on the application of the tool for structural adaptations for the case of Teleservices and Remote Medical Systems. Similarly Section 5.7 reports on the application of RTF4ADS for behavioral adaptations for the case of Toast Architecture. Finally, Section 5.8 summarizes the chapter.

5.2 RTF4ADS overview

Getting confidence in dynamic and distributed software systems can be reached by using RTF4ADS as a resource aware and platform independent test support. On the one hand, resource awareness is achieved by distributing selected tests according to available resources and connectivity constraints of the final execution nodes. On the other hand, platform independence is reached using the TTCN-3 standard. The latter provides a text-based language that

inherits the most important programming features and includes some specific concepts related to the testing domain. Its strength lies essentially in its reference test architecture that automates test execution and more particularly in its test adaptation layer. The latter comprises Coding-Decoding entity, Test Adapter entity and Platform Adapter entity that supply means to adapt the communication and the time handling between the SUT and the test system in a loosely-coupled manner.

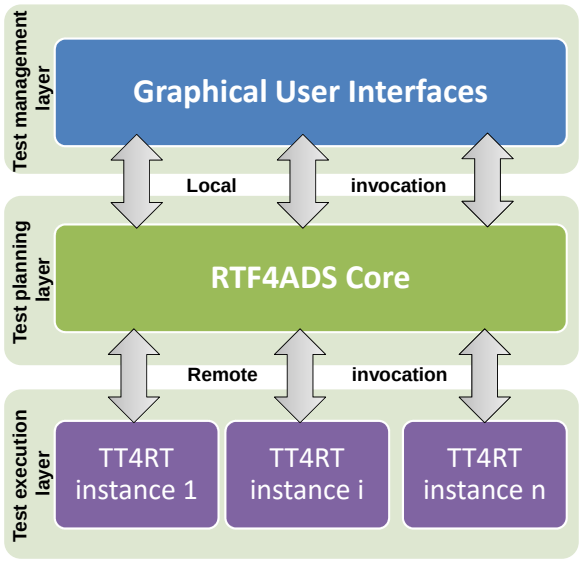


Figure 5.1: RTF4ADS prototype.

As depicted in Figure 5.1, this Java-based framework comprises three layers :

- **Test management layer:** Graphical User Interfaces (GUI) are provided to handle automatically the different phases of the runtime testing process.
- **Test planning layer:** The RTF4ADS core includes modules that contribute efficiently to the test generation, the test selection and the test distribution steps.
- Test execution layer: Several TT4RT instances are deployed and charged with first applying test isolation mechanisms and second executing runtime tests.

In the following, we introduce each GUI while presenting its corresponding involved modules.

5.3 Test selection and distribution GUI

The GUI component, illustrated in Figure 5.2, is used by the Test System Coordinator to plan the execution of runtime tests. It is responsible for analyzing SUT dependencies, selecting test cases to rerun and looking for a test component placement solution for the involved main test components while fitting resource and connectivity constraints. More details are given next.

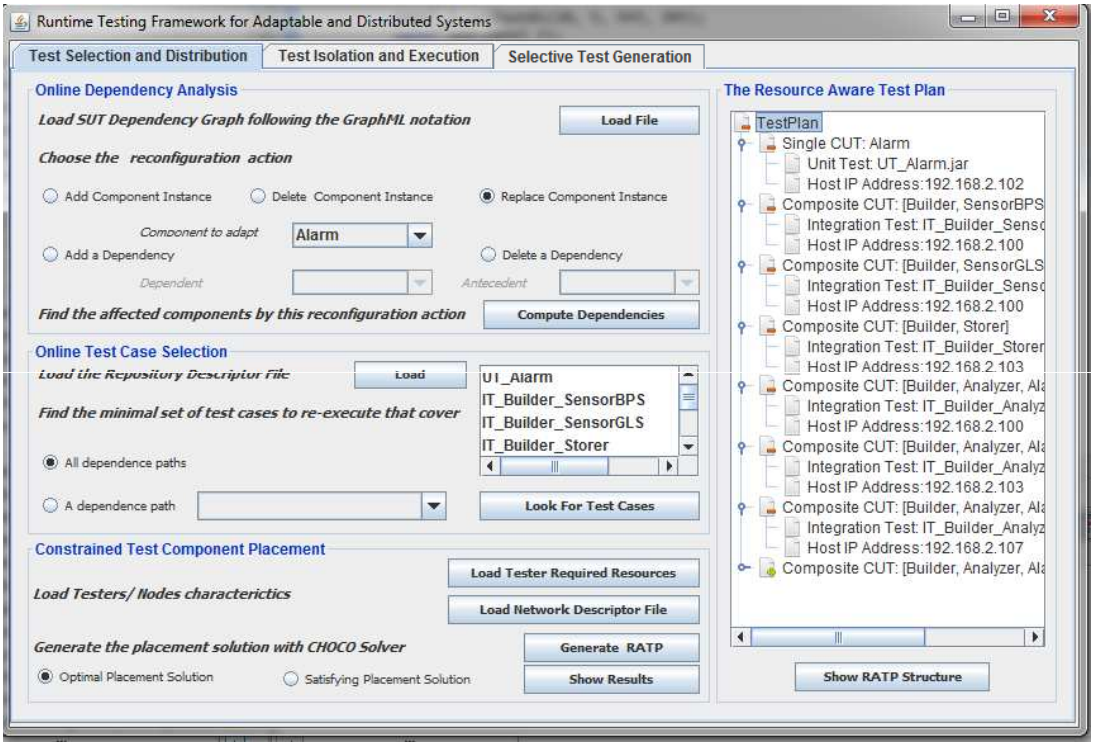


Figure 5.2: Screenshot of the test selection and distribution GUI.

- **Dependency Analysis Module:** It takes as inputs the performed reconfiguration action and a file that describes the system dependency graph. The latter is expressed in the *Graph Markup Language* (GraphML). Indeed, GraphML notation (98) is an XML-based file format for graphs. It consists of a language core to describe the structural properties of a graph and a flexible extension mechanism to add application-specific data.

- **Test Case Selection Module:** It requires two major inputs. The first input is the Test Case Repository Description that expresses, for each test stored in the repository, data like identifiers, names, artifacts, MTC components, required resources, etc. The second input is the set of affected components and compositions obtained from the last step. The main goal at this stage is to look for a minimal set of test cases to run.

- **Tester Placement Module:** It is used to distribute TTCN-3 tests and their corresponding MTC components to the execution nodes while respecting already defined resource and connectivity constraints. The core of this module is based on the Choco Java library which is an open source software offering a problem modeler and a constraint programming solver (99). The generated output is a Resource Aware Test Plan.

5.4 Test isolation and execution GUI

The second component GUI, depicted in Figure 5.3, is used by the Test System Coordinator to start remotely one or several tests. For this purpose, it communicates with several TT4RT instances by using the *Remote Method Invocation* (RMI) technology. It also displays the global verdict, local verdicts collected from each involved host in the runtime testing process and some logging data.

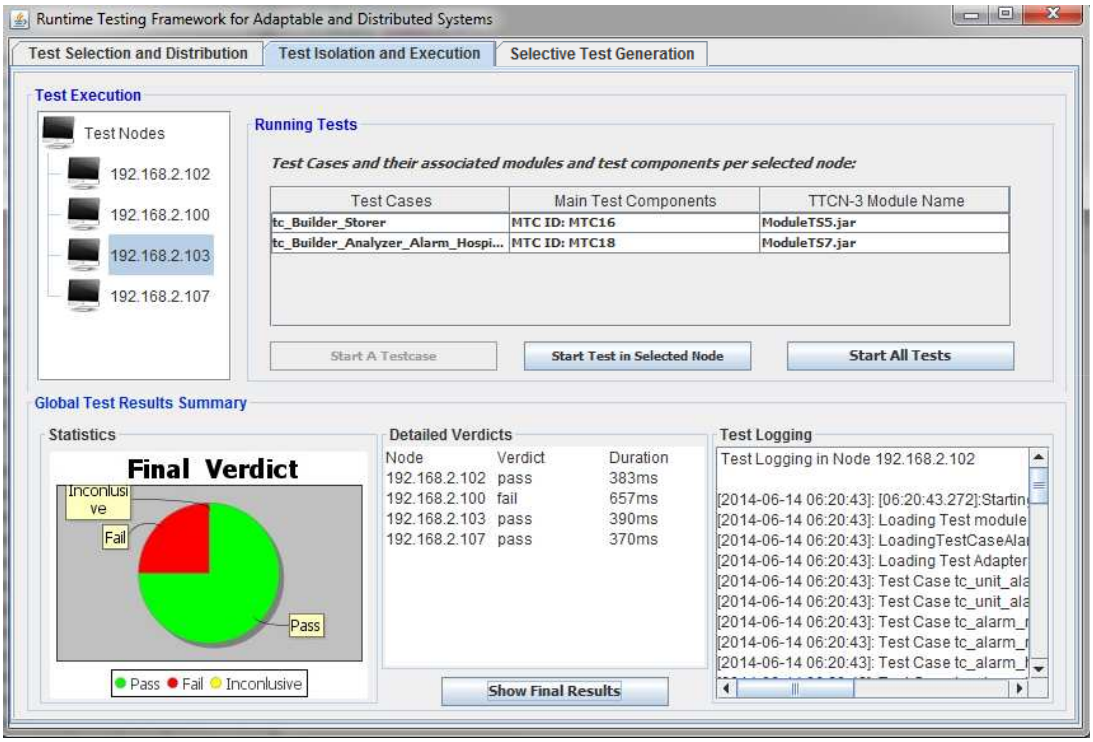


Figure 5.3: Screenshot of the test isolation and execution GUI.

The first JTree panel outlines the involved nodes in the test execution process. In each one, a TT4RT instance is installed and started. Two majors input elements are required by TT4RT : selected Executable TTCN-3 test cases from the repository as JAR files and the Resource Aware Test Plan.

The centered JTable describes test cases assigned to the selected node as well as their main characteristics (i.e., test case name, TTCN-3 module name and MTC identifier). Several buttons are proposed to efficiently manage the test execution. Consequently, we can start a selected test case, all tests in a selected node or even all tests on their corresponding nodes. In that case, each TT4RT instance is designed as a remote server object which implements a remote interface offering three methods.

The *Generic Test Isolation Component*, which represents the test isolation layer in TT4RT, implements a *User Datagram Packet* (UDP) port listener function which runs an infinite loop

listening for test data in the form of UDP packets from the UDP test adapter. It can intercept either local test requests sent by a local test adapter or remote test requests sent by a remote test adapter. It is worth noticing that in the current implementation of RTF4ADS, not only a UDP test adapter was implemented but also a *Transmission Control Protocol* (TCP) test adapter was encoded. The latter can be easily integrated if required. These two possible implementations can be used to establish communication through sockets between our TS and any kind of SUT.

This component uses AOP facilities to automate the test isolation of components under test before the execution of runtime tests. In fact, we associate for each provided interface a test isolation instance, designed as an AOP advice, which is automatically launched if at least one of its methods is called by a test component. This test isolation instance is charged with looking for the testability option of the component under test and then proceeds to the test execution. To realize such an implementation, we use the most popular and stable AOP language, namely AspectJ (100). Indeed, the latter extends the Java language with new features to support the aspect concepts.

5.5 Selective Test Generation GUI

The RTF4ADS framework includes a selective test generation GUI to build automatically a new test suite after the occurrence of behavioral adaptations.

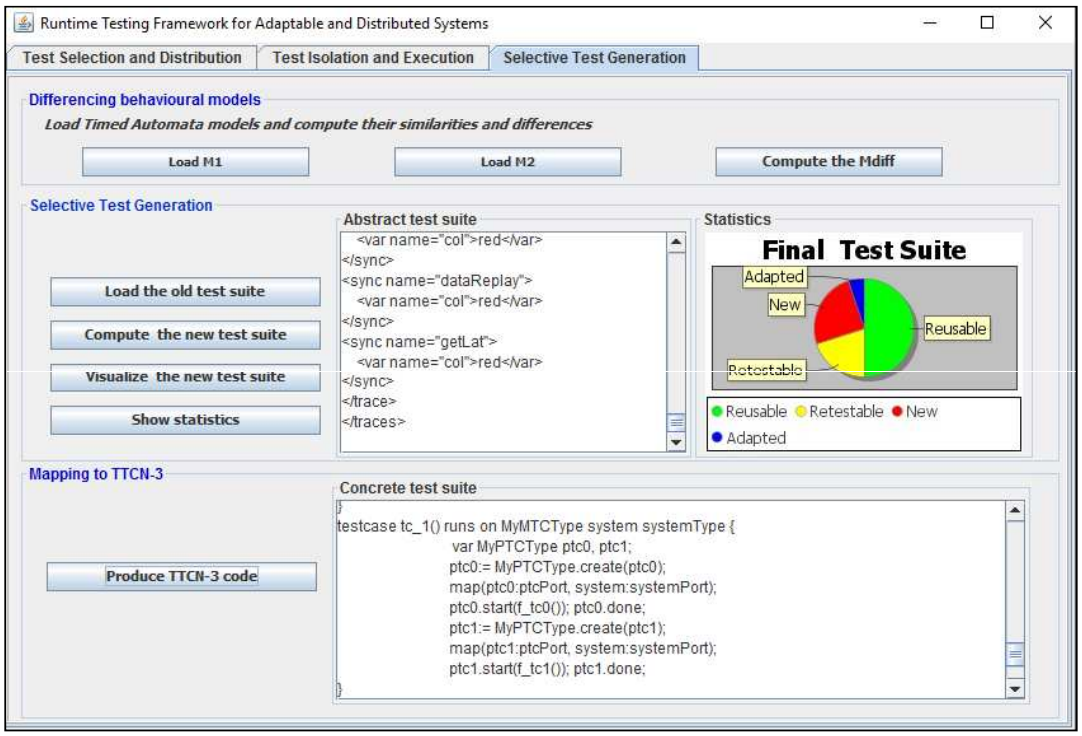


Figure 5.4: Screenshot of the selective test generation GUI.

The new test suite is composed of reusable and retestable tests, selected from the old test suite, new tests generated from the evolved behavioral model by UPPAAL CO$\sqrt{}$ER (version 1.4) (87) and some adapted ones obtained by test recomputation. Once the new test suite is evolved, it is mapped to the TTCN-3 notation.

The first panel illustrated in Figure 5.4 deals with the model differencing step. Indeed, the initial behavioral model and the evolved one are loaded, and then an M_{diff} model highlighting their similarities and their differences is computed. The next step consists in loading the old test suite and performing a test classification.

Finally, we compute the new test suite by launching the UPPAAL CO$\sqrt{}$ER tool to generate new tests, in a cost effective manner, by adapting obsolete and aborted tests and by including resuable and restestable tests.

5.6 Application of RTF4ADS for Structural Adaptations

5.6.1 Teleservices and Remote Medical Care Systems

Teleservices and Remote Medical Care Systems (TRMCS) were introduced in the literature for more than a decade ago (101). They were designed initially to provide monitoring and assistance to patients suffering from chronic health problems. Thus, they send emergency signals to the medical staff (such as doctors, nurses, etc.) to inform them with the critical state of a patient. Recently, both the architecture and the behaviors of such medical care systems are evolved and enhanced by more elaborated functionalities (for instance, the acquisition, the analysis and the storage of biomedical data) and sophisticated technologies (102; 103; 5; 6).

New components and features can be installed at runtime during system operation in order to fulfill new requirements such as adding new health care services, updating the existing ones in order to support performance improvements, etc. Such adaptability is essential to ensure that the healthcare system remains within the functional requirements defined by application designers, and also maintains its performance, security and safety properties.

Following these directions, we provide our own architecture of the TRMCS application which is inspired mainly from (103). Its main architecture is highlighted through a UML deployment diagram depicted in Figure 5.5. We assume that initially a given patient is suffering from chronic high blood pressure. Thus, he is equipped with a *Blood Pressure Sensor* and a *Heart Rate Sensor* that measure respectively his arterial blood pressure and his heart-rate beats per minute. Periodic reports are built and stored in the medical database. They are also accessible for consultation from the medical staff. The *Analyzer* component is charged with analyzing the

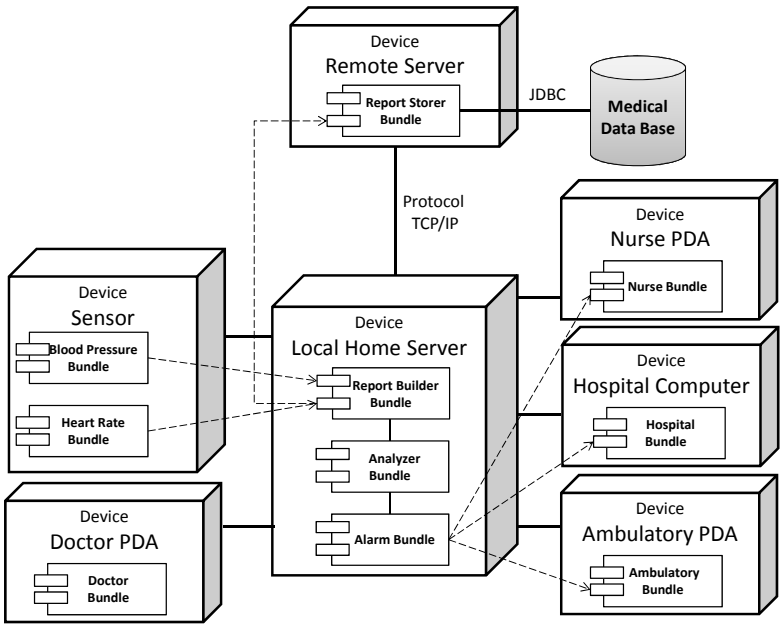


Figure 5.5: The basic configuration of TRMCS.

monitored data in order to detect whether some thresholds are exceeded. In that case, an *Alarm* component is invoked with the aim of sending help requests to the medical staff.

5.6.2 TRMCS implementation

The TRMCS is fully implemented as an OSGi application. Developed by OSGi Alliance (15), the OSGi specification describes how to build service-oriented and loosely coupled systems. This standardized technology defines a lightweight framework based on Java Runtime Environment and a set of installed *bundles*. Bundles are software components, packed in JAR files. A bundle is designed as a minimal deliverable application in OSGi that is composed of cooperating services, which are discovered after being published in the OSGi service registry. It is capable of either exporting Java packages and exposing functionalities as services to other bundles or importing Java packages or services from other bundles (104; 105).

5.6.3 TRMCS test specification

To show the high expressiveness of the TTCN-3 language in supporting various testing levels (i.e., unit and integration tests) and different testing purposes (i.e., functional tests, load tests, availability tests, etc.), some test scenarios are studied and their mapping to the TTCN-3 notation is given afterwards. Given that the entire test scenarios are too lengthy to describe, four examples are provided to highlight the most common types of test scenarios. First of all, these scenarios are introduced in a descriptive way then their mapping to the TTCN-3 code is given.

- **Guarantee of help service delivery:** This scenario can be used to test the situation in which the analysis of monitored critical events are triggered or threshold conditions are reached (i.e., when the heart rate exceeds a certain level of tolerance). In this context, emergency signals are sent to the appropriate medical staff.

- **Achievement of timing constraints:** This scenario is used to check that the *Alarm* component must send the help request to the *Nurse* component in a duration that does not exceed 15 time units.

- **Availability of a component:** This scenario serves to check component availability after the occurrence of dynamic reconfigurations (i.e., adding, updating or migrating components). For instance, in case of patient mobility in and out of the local server's range, we have to check that wearable medical sensors are accessible and can be invoked from components deployed on the local server.

- **Concurrent test requests:** This scenario is used to simulate the situation in which multiple users request the service under test at the same time. The dynamic creation of PTCs in TTCN-3 standard enables our framework to create a number of virtual users that send multiple test requests concurrently and perform load testing on the SUT.

In order to edit and compile the specified tests, we use respectively the *TTCN-3 Core Language Editor* (CL Editor) and the *TThree Compiler* that are included in the TTworkbench basic tool. The generated Jars are stored in the Test Case Repository for further use and can be dynamically loaded during the runtime test execution to check dynamic changes.

5.6.4 Evaluation and overhead estimation

We carried out some experiments to measure the overhead introduced by the use of the RTF4ADS framework when structural adaptations take place. Thus, the main objective is to estimate the dependency analysis, the test selection, the constrained test component placement and the test execution overheads and to determine which parameters have a significant effect on each of them. Thereby, we deployed our distributed test system as well as the TRMCS application on five machines: a PC with Intel Core 2 Duo CPU and 2 GO of main memory, another PC with Intel Core i7 and 8 GB of main memory and three virtual machines having each 2.30 GHz CPU and 512 MB of main memory. Using this experimental setting, we deployed four TT4RT instances on the involved test nodes identified during the test component placement step. RTF4ADS user interfaces were deployed on a separate host (see Figure 5.6).

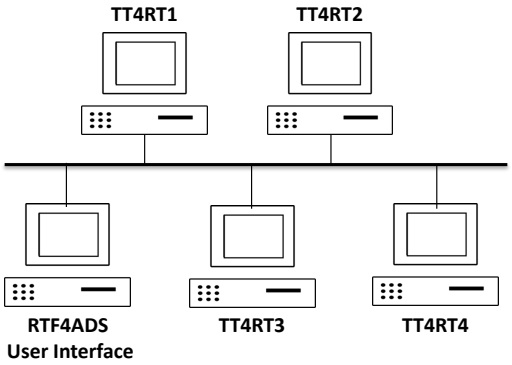


Figure 5.6: The adopted testbed.

The different experiments that we carried out show that the runtime testing cost in terms of execution time and memory consumption increases significantly while the amount of tests to run or the number of test components to deploy rises. Compared to one of the traditional test selection strategy, the *Retest All* strategy (106), which re-executes all available tests, our proposal seems to be more efficient as it reduces the number of tests to rerun. In addition, the adopted test selection technique does not require much time to identify the unit and integration tests involved in the runtime testing process. Even in the worst case, when the whole system is affected by the dynamic change, we reduce the impact of the runtime testing on the system under test and on its environment by distributing test cases and their corresponding test components while fitting the resource and connectivity constraints.

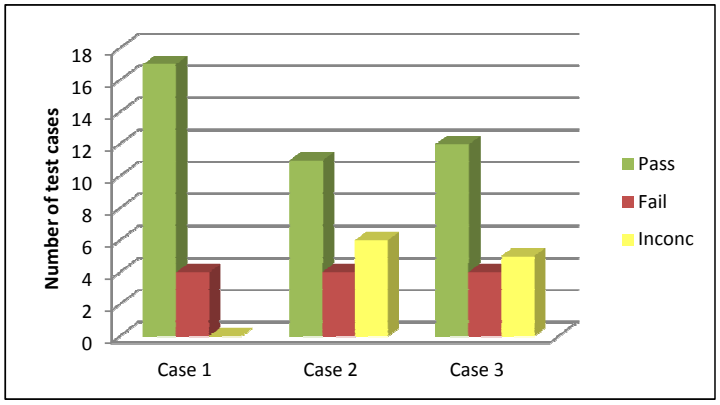


Figure 5.7: The impact of resource and connectivity awareness on test results.

The experiment outlined in Figure 5.7 shows the importance of the constraint test component placement module and its impact on the final test results. In the following, three cases of test results are obtained while executing twenty one selected tests.

- In the first case, our test placement module was used to identify the adequate test hosts and our test system was run to perform the selected tests. The seeded faults were detected and thus we obtained **seventeen** Pass and **four** Fail verdicts.

- In the second case, we assume that the hospital computer is disconnected from the network. However, this connectivity problem was not taken into consideration during the testing process. As a result, six test requests were sent from their corresponding test components without receiving any response. In this situation, neither a Pass nor a Fail verdict can be assigned and thus **six** inconclusive verdicts are obtained.

- The third case shows the test results obtained while executing the twenty one tests on some overloaded nodes. As in Case 2, the tests results are influenced by the execution environment state and consequently several verdicts were set to inconclusive. Such test results were obtained due to the timeout occurrence during the test execution.

To sum up, runtime testing may affect not only the SUT performance and responsiveness but also the test system itself could be impacted. Thus, resource and connectivity awareness appears to be a solution in order to have a high confidence in the validity of the test results as well as to reduce their associated cost.

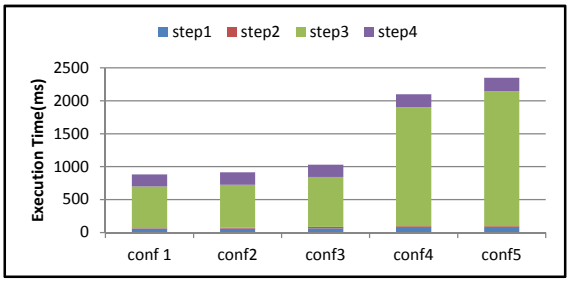


Figure 5.8: The overhead of the whole runtime testing process while searching for an optimal solution in step 3.

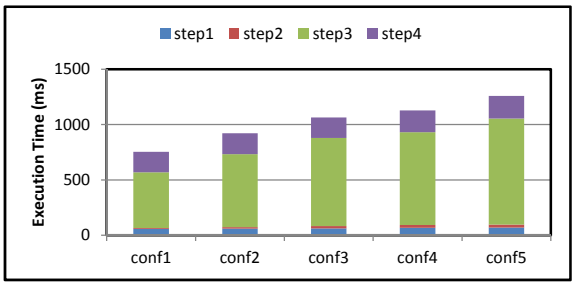


Figure 5.9: Assessing the overhead of the whole runtime testing process while searching for a satisfying solution in step 3.

The experiments presented in Figure 5.8 and Figure 5.9 show that the different overheads introduced by our runtime testing support are relatively low. In fact, we find out that the sum of all overheads including the dependency analysis (step 1), the test selection (step 2), the test placement (step 3) and the test execution (step 4) overheads does not exceed 2.5 seconds in the worst case (i.e., when we are looking for an optimal solution in step 3). This cost can be justified by the use of exact methods in the current version of the constrained test component module. It is obvious that this resolution technique is one of the most costly ways to find the best solution from all feasible solutions. As illustrated in Figure 5.9, this cost decreases when we simply look for a feasible solution of test component placement. In this case, our test framework requires less than 1.5 seconds for checking *Conf 5*. With the aim of guaranteeing the scalability of RTF4ADS, we recommend to make do with generating the satisfying solution as it consumes less time when the numbers of test components and host nodes increase.

5.7 Application of RTF4ADS for Behavioral Adaptations

5.7.1 Toast architecture

The interest in telematics and fleet management systems has witnessed an increase during the last decade. The first generation of these systems provides simple functionalities such as vehicle tracking systems. The latters include but they are not limited to the *Global Positioning System* (GPS) technology integrated with other advanced sensors and the mobile communication technology. Currently, fleet management systems are more and more mature and highly developed. Consequently, they involve sophisticated functions such as the supervision of the use and the maintenance of vehicles, the monitoring and the accident investigation capabilities, and so on. Moreover, the flexibility and the dynamic adaptability have become important attributes of a fleet management system with the aim of adapting its behavior to the changing needs of the industry and the increasing evolution in the automotive area.

Seeing all these features, a sample case study in this emergent domain is retained to show the feasibility of our selective test generation approach after the occurrence of behavioral adaptations. As introduced in (107), Toast is a typical fleet telematics system used to demonstrate a wide range of EclipseRT technologies. As an OSGi-based application, it provides means to manage and to interact with vehicle devices at runtime. Initially, we start with a simple scenario that covers the case of emergency notification. In this situation, the vehicle comprises three devices : an Airbag, a GPS and a Console. If the airbag deploys, an Emergency Monitor is notified. The monitor asks the GPS for the vehicle position and speed (see Figure 5.10) and displays the obtained data on the vehicle console.

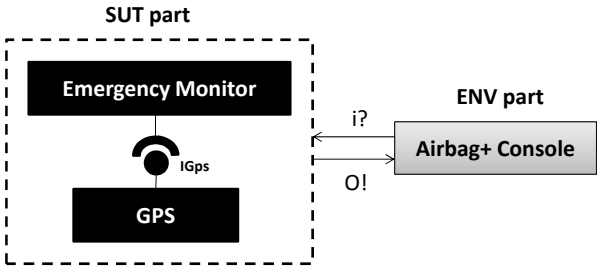
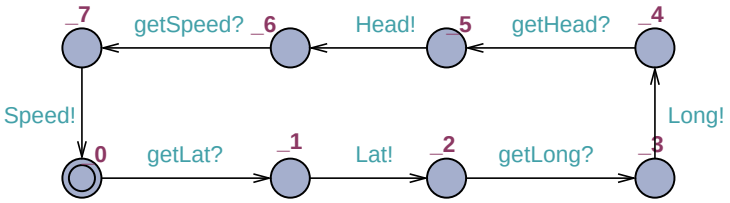


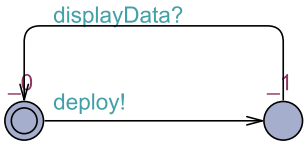
Figure 5.10: The initial Toast architecture.

In the following, we consider the Airbag and the Console components as a part of the environment. Our system under test is made up of GPS and Emergency Monitor components. SUT and ENV parts are modeled by a network of UPPAAL timed automata as shown in Figure 5.11. At the beginning, timing constraints are not considered and we focus mainly on the synchronization of input and output signals between the Toast components.

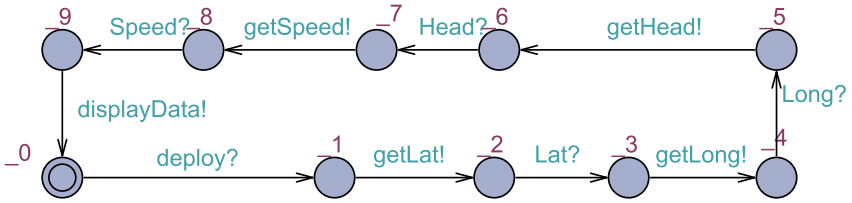
When the Airbag is deployed (via the action *deploy*), the Emergency Monitor interacts with the GPS to get the vehicle's latitude, longitude, heading and speed. Once this information is obtained, it is displayed on the console with an emergency message (modeled by the action *displayData*).



(a) The initial GPS model.



(b) The environment model.



(c) The initial Emergency Monitor model.

Figure 5.11: Toast behavioral models.

5.7.2 Dynamic Toast evolution

Starting from the basic configuration introduced in the previous section, new components and features can be installed at run-time during the system execution. For instance, we can add a new application that tracks the vehicle's location and periodically reports to the control center. A support for climate control can be integrated, as well. As illustrated in Table 5.1, six cases of behavioral adaptations are considered.

5.7.3 Applying the selective test generation method after Toast evolution

To check the correctness of the evolved Toast application in a cost effective manner, we have to evolve the test suites by making use of the TestGenApp module. The first step in this module consists in comparing the initial behavioral model and the evolved one. As output, it generates an M_{diff} model that highlights the similarities and difference between timed automata. It is worthy to note that several cases of evolution are studied in the following. For each case, the obtained M_{diff} model is automatically exported from the UPPAAL model checker. Once the test generation process is achieved, the transformation of the abstract test sequences to concrete tests should be performed.

Table 5.1: Several studied Toast evolutions.

Scenarios	Evolution description	Kind of the evolution	SUT templates	Locations	Transitions
Case 0	Initial Toast configuration	———	GPS Emergency	8 10	8 10
Case 1	Updated GPS behavior	Complex (adding locations and transitions)	GPS Emergency	13 15	15 17
Case 2	Error support in GPS data transmission	Complex (adding locations and transitions)	GPS Emergency	27 29	36 38
Case 3	Removal some behaviors within the GPS	Complex (removing locations and transitions)	GPS Emergency	23 25	31 33
Case 4	Addition of the Back End Server	Complex (adding a new template)	GPS Emergency Back End	23 26 3	31 34 3
Case 5	Addition of the Tracking Monitor	Complex (adding a new template)	GPS Emergency Tracking Back End	23 26 11 6	31 34 11 6
Case 6	Addition of the Climate Controller and the Climate Monitor	Complex (adding two new templates)	GPS Emergency Tracking Back End Climate Monitor Climate Controller	23 26 11 6 9 6	31 34 11 6 12 9

5.7.4 Test distribution and execution

At this stage, the new abstract test suite is computed after the occurrence of behavioral adaptations. In addition, its mapping to the TTCN-3 notation is achieved with the aim of obtaining concrete tests. Once the latter are compiled by using the TTthree compiler, executable TTCN-3 tests are ultimately produced. To execute the obtained tests, RTF4ADS is called, more concretely its constraint test placement module as well as its test isolation and execution module.

5.7.5 Evaluation and overhead estimation

In this section, we carried out some experiments to measure the overhead introduced by the use of TestGenApp module when different scenarios of behavioral evolutions take place. Thus, the main objective is to compute the number of generated traces after each evolution and estimate the execution time required for the model differencing step, the test classification step and ultimately for the test generation step with UPPAAL CO$\sqrt{}$ER.

Table 5.2 illustrates the studied Toast evolution scenarios and pinpoints the comparison between our proposal TestGenApp and two well-known regression testing strategies : the *Regenerate All* and the *Retest All* approaches. Recall that the first one consists in generating all tests from the new evolved model. The second approach deals with re-executing all tests in the old test suite issued from the old behavioral model and generating new tests that cover only new added behaviors.

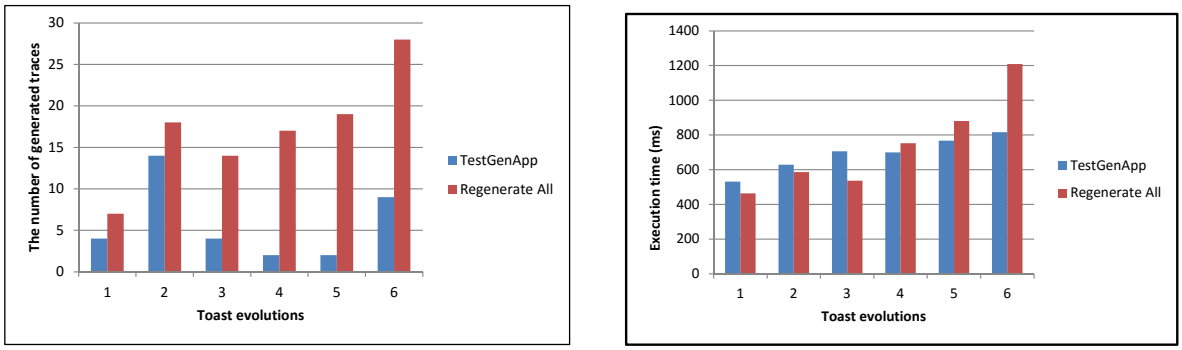
Table 5.2: Comparison between Regenerate All, Retest All and TestGenApp strategies.

Scenario	Case study evolutions	Regenerate All	Retest All		TestGenApp			
			Old	New	Reusable	New	Retestable	Adapted
1	From Case 0 to Case 1	7 traces	3	4	1	4	2	0
2	From Case 1 to Case 2	18 traces	7	14	1	14	6	0
3	From Case 2 to Case 3	14 traces	18	0	1	0	11	4
4	From Case 3 to Case 4	17 traces	14	2	7	2	7	0
5	From Case 4 to Case 5	19 traces	17	2	17	2	0	0
6	From Case 5 to Case 6	28traces	19	9	19	9	0	0

Compared to the Regenerate All technique, our proposal reduces the number of generated traces as shown in Table 5.2. For instance, the evolution from Case 0 to Case 1 requires the generation of **seven** traces with the Regenerate All strategy. The application of TestGenApp produces the selection of **one** trace as a reusable test that covers the unimpacted parts of the model. Moreover, **two** old traces are classified as retestable tests. Only **four** traces are newly generated to cover the newly-added transitions in both the GPS and the Emergency models.

Similarly, instead of generating the full test suite (**fourteen** traces here) when the Toast architecture evolves from Case 2 to Case 3, only **four** traces are adapted in order to cover the modified transitions in the SUT models. **Eleven** old traces are still valid and can be re-executed to prove that these reductive changes have no side effects on the unimpacted parts of the model. Moreover, **one** trace is considered as a reusable test.

Concerning the Retest All strategy, we notice that this strategy does not make any analysis before re-executing tests. Its main limitation consists in re-executing obsolete tests which are no longer valid. For example, when the Toast evolves from Case 2 to Case 3, **four** traces from the old test suite cannot be animated on the new model and then they may cause failure during test execution. This failure is not caused by a faulty behavior in the system but it is due to the execution of invalid tests. Consequently, we conclude that selecting valid and relevant tests to run is highly recommended because it provides a high degree of confidence in the evolved system without rerunning the overall test suite.



(a) The number of generated traces.

(b) Execution time for test evolution.

Figure 5.12: Comparison between TestGenApp and Regenerate All approaches.

Figure 5.12 outlines two experiments that we conducted on a machine with Intel Core i7 and 8 GB of main memory. They show that TestGenApp and Regenerate All approaches depend highly on the model scale either in terms of generation time or generated traces.

Regarding the number of generated traces after each evolution, we notice that an increase in the number of involved templates, locations and transitions causes an increase in the test suite size. As depicted in Figure 5.12a, it is obvious that the TestGenApp produces less traces than the Regenerate All strategy since it focuses only on covering new behaviors in the evolved model.

Regarding the generation time, Figure 5.12b shows that this measure follows the model scale, as well. In case of small systems (e.g., Toast scenarios in Case 1, Case 2 and Case 3), TestGenApp overhead in terms of test generation time is greater than Regenerate All as it performs several tasks : model differencing, test classification and test generation (see Figure 5.13). When we deal with large systems, we notice that the cost of generating the complete test suite is higher than generating only new behaviors.

Figure 5.13: The overhead of the TestGenApp modules.

Such experiments show the clear benefits of the TestGenApp especially in case of large scale models and elementary modifications (i.e., Adding/removing/modifying a location and/or a transition). It is easier to generate a minimal set of tests from the part of the model impacted by the dynamic change rather than performing a full regeneration.

Compared to the typical solutions such as Regenerate All and Retest all, TestGenApp gives an important information about the obtained tests and which parts of the SUT they cover. As a result, test prioritization can be easily applied in our context and tests covering critical zones have the priority to be executed first.

5.8 Summary

The achievement of a well-implemented prototype for runtime testing of dynamic adaptations was pinpointed in the present chapter. The obtained framework includes the realization of both approaches proposed to check structural and behavioral adaptations. Some implementation details in terms of input files, output results and used tools required for each module were presented. Moreover we reported on two case studies. On the one hand, we illustrated the efficient execution of runtime tests in case of dynamically adapting the structure of an e-Health case study. On the other hand, our tool was used to update test suites when behavioral adaptations take place in the case of a telematic application.

Part II

Combining Load and Functional Tests

CHAPTER 6

A Comparative Evaluation of State-of-the-Art of Load Testing Approaches

6.1 Introduction

In order to deliver quality assured software and avoid potential costs caused by unstable software, testing is essential in software life cycle. Load testing is one of the testing types with high importance. It is usually accompanied by performance monitoring of the hosting environment. The purpose of this chapter is to present related solutions to load testing issues in different fields and emerging paradigms, and to evaluate them in order to identify their advantages and their shortcomings. Looking at the areas focused by existing researchers, gaps and untouched zones of different systems relatively to load testing can be discovered. This investigation is a preliminary step for our research work in the context of load testing of Web service compositions, considered as an arising concept in Service-Oriented Architecture (SOA).

The remainder of this chapter is organized as follows. Section 6.2 explains the motivation behind the work presented in this chapter. Some background information on load testing concepts and a general overview of their challenges are presented in Section 6.3. A classification of the load and stress testing approaches is presented in Section 6.4. Section 6.5 lists the criteria used in the comparative evaluation as well as a summary of the evaluation. Finally Section 6.6 concludes the chapter.

6.2 Motivation

Many software applications must provide services to hundreds or thousands of users concurrently. These applications must be load tested to ensure that they can function correctly under high load. Indeed, load testing is a process of subjecting a computer, peripheral, server, network or application to a workload approaching the limits of its specifications. Unlike stress testing, which evaluates the extent to which a system keeps working when subjected to extreme workloads or when some of its hardware or software has been compromised, the primary goal of load testing is to define the maximum amount of work a system can handle without significant performance degradation. For that, load testing requires one or more load generators which mimic clients sending thousands or millions of concurrent requests to the application under test. During the course of a load test, the application is monitored and performance data along with execution logs are stored.

However, load testing is usually very time consuming. A test run typically lasts for several hours or even a few days. A memory leak, for example, might take hours or days to fully manifest itself. Therefore, load tests have to be long enough. Also, load tests are usually performed at the end of the testing phase after all the functional tests. Testing in general is performed at the end of a development cycle. Thus, load testers are usually under a tremendous pressure to finish testing and certify the software for release.

In this context, several approaches have been proposed with the aim to perform load or/and stress testing of different systems, such as program codes (108), network applications (109), distributed systems (110) and software applications (111), (112). This chapter focuses on load and stress testing issues in general and offers a detailed survey of state-of-the-art testing approaches. By offering an overview and a classification of current approaches which are related to both load and stress testing, it is hoped to provide an essential outlook for future research in different areas, particularly concerning load testing of Web service compositions, both in academia and industry.

6.3 Load & Stress Testing

In this section, we present some related concepts and challenges to both load and stress testing areas.

6.3.1 Definitions

Testing is an important stage in software life cycle. It is an assurance of software quality and an effective way to avoid potential costs caused by unstable software. Indeed, it verifies the expected results are achieved or not and correct the bugs as soon as possible. In software development processing, bugs always exist no matter what technology is adopted. Thus, testing is applied to find bugs, and used to calculate software bugs density (113). In a typical software project, the percentage of software testing workload is about 40%. Particularly, many systems ranging from e-commerce websites to telecommunications must support concurrent access by hundreds or thousands of users. To assure the quality of these systems, load testing is a required testing process in addition to conventional functional testing procedures, such as unit and integration testing, which focus on testing a system based on a small number of users. In fact, load and stress testing are important to guarantee the system is able to support specified load conditions as well as properly recover from the excess use of resources. The generation of test cases to achieve levels of load and stress is thus a demanding task.

On the one hand, *load testing* (114; 115) assesses how a system performs under a given load. The rate at which transactions are submitted to the system is called the *load* (116). Load generators are used to induce load on the system under test (117), i.e., imitating thousands of users committing concurrent transactions to a system. During the course of a load test, the system is strictly monitored and important sources of data exposed by the system, i.e. performance metrics, are collected. These metrics include numerical measurements related to the system's state and performance (e.g., CPU, memory utilization, network usage, etc.). One of load testing objectives is to determine the maximum sustainable load the system can handle. It reveals programming errors that would not appear if the system executes with a small/limited workload. Such errors are called *load sensitive faults* and emerge when the system is executed under a heavy load.

On the other hand, *stress testing* (116; 114; 115) refers to subject a system to an unreasonable load with the intention of breaking it. A stress test denies a system the resources (e.g., RAM, disk, interrupts, etc.) needed to process a certain load. It is designed to cause a failure and to test the system's fault recovery capability. The system is not expected to process the overload without adequate resources, but to behave (e.g., fail) in a reasonable manner (e.g., not corrupting or losing data). By automatically driving the resource usage to its limit, load sensitive faults can be detected and performance issues can be verified under stress conditions. The automatic identification of these faults can have a large impact on the quality of the products released as well as on the reduction of the required test effort.

6.3.2 Challenges

Load testing is an area of research that has not been explored much. Actually, load testing is a difficult task requiring a great understanding of the application under test. Besides, load testing involves the setup of a complex load environment. The application under test should be setup and configured correctly. Similarly, load generators must be configured properly to ensure the validity of the load test. Then, test results must be analysed closely to discover any problem in the application under test, in the load environment, or in the load generation.

Particularly, load testing uncovers residual functional and performance problems that slipped through the conventional functional testing. Indeed, a *functional problem* results in processing happening at the wrong place in the wrong order (118). In order to verify functional correctness, load test engineers first check whether the application has crashed, restarted or hung during the load test. Then, they perform a more in-depth analysis by grepping through the log files for specific keywords like *failure* or *error*. Load test engineers analyse the context of the matched log lines to determine whether these lines indicate problems or not. There are two limitations in the current practice for checking functional correctness. Firstlym terms like 'error' or 'failure' are worth investigating. A log such as *Failure to locate item in the cache* is probably not a bug. Secondly, not all errors are indicated in the log file using the terms 'error' or 'failure'.

On the other hand, *performance problem* results in processing taking too much or too little of important resource. A request that takes too long may indicate a bottleneck, while a request that finishes too quickly may indicate truncated processing or some other performance bug (112). With the aim to evaluate performance criteria, load test engineers first use domain knowledge to check the average response time of a few key scenarios. Then, they examine performance metrics for specific patterns. Finally, they compare these performance data with previous releases to assess whether there is a significant increase in the utilization of system resources. However, the current performance analysis practice is not efficient, since it takes hours of manual analysis. Current practice is neither sufficient for the following two reasons: first, checking the average response time does not provide a complete picture of the end user experience, as it is not clear how the response time evolves over time or how response time varies according to load. Second, merely reporting symptoms like *system is slowing down* or *higher resource utilization* does not provide enough context for developers to reproduce and diagnose the issues.

All the previously described challenges may explain the existence of relatively few works dealing with load and stress testing in various fields as introduced in the next section.

6.4 Classification of Load & Stress Testing Solutions

Many systems must offer services to a great number of users at the same time. Thus, they must be load tested in order to guarantee that they behave correctly under high load. In this section, we expose some related solutions to both load and stress testing issues in different fields.

6.4.1 Load & Stress Testing Tools

Load testing tools are used for software performance testing in order to create a workload on the system under test, and measure response times under this load. Load testing tools are available from large commercial vendors such as Borland, HP Software, IBM Rational and Web Performance Suite, as well as Open source projects. In the following, we expose efforts made particularly in the context of Web applications and Web service compositions load testing.

- **Web Applications Field:** Nowadays, Web applications are widely used, one fact is obvious: most of Web applications are public and used by vast number of users, which are making a considerable traffic load on hosting environments and Web sites. By the way, (119) analysed and compared several existing tools which facilitate load testing and performance monitoring, in order to find the most appropriate tools by criteria such as ease of use, supported features, and license. Selected tools were put in action in real environments, through several Web applications. In order to introduce different capabilities of tools, including distributed testing, security support, results analysis, monitoring of key parameters, etc., (119) presented the test results of a Web application being developed in ENT (Ericsson Nikola Tesla). They concluded at the end that concerning load testing tools, the most required feature was support for performing load test process steps, with emphasis on recording, distributing tests, HTTPS and AJAX support.

- **Web Service Compositions Field:** Furthermore, Web service compositions provide services to thousands of users concurrently. These applications must be load tested to ensure that they can function properly under high load. In this context, the infrastructure of Oracle BPEL Process Manager, an existing commercial tool, offers a solution for deploying and managing designed BPEL processes in particular. Moreover, the BPEL console, which is provided by Oracle BPEL Process Manager, includes stress test capability that makes it possible to perform load testing of a deployed BPEL process and then to view performance statistics of the flow under stress. Enabling stress test permits to perform a continuous series of invocations of the Web service operation. At the end of test, Oracle BPEL Process Manager generates a final report.

6.4.2 Load & Stress Testing Approaches

There are different research works dealing with load and stress testing in various contexts. In the following, we describe these approaches in chronological order.

- **Avritzer (1993, 1994, 1995):** (120; 121),(122) introduced a load testing technique called Deterministic Markov State Testing. This approach is limited to applications that can be modeled by a Markov Chain since the input data is assumed to arrive according to a Poisson distribution and is serviced in an exponential distribution. In addition, the authors proposed a class of load test case generation algorithms for telecommunication systems which can be modeled by Markov chains. The proposed black-box techniques are based on system operational profiles. Indeed, the operational profile is used to build a Markov chain that represents the software's behaviour. Only the most likely test cases, as computed from the most probable Markov chain states, are generated at planning time; i.e. before the start of system test. Each test case certifies a unique software state. For that, the Markov chain is first built. The operational profile of the software is then used to calculate the probabilities of the transitions in the Markov chain. The steady-state probability solution of the Markov chain is then used to guide the generation process of the test cases according to a number of criteria, in order to target specific types of faults. From a practical standpoint, targeting only systems which behaviour is modeled by Markov chains can be considered as a limitation of this work. Furthermore, using only operational profiles to test a system may not lead to stressing situations.

- **Yang (1996):** (108) proposed a technique to identify the load sensitive parts in sequential programs based on a static analysis of the code. A load sensitive part is defined as a part the correctness of which depends on the amount of input data or the length of time that the program will execute continuously. In addition, the authors illustrated some load sensitive programming errors, which may have no damaging effect under small loads or short executions, but cause a program to fail when it is executed under a heavy load or over a long period of time. Their proposed technique targets memory-related faults (e.g., incorrect memory allocation/de-allocation, incorrect dynamic memory usage) through load testing. To explain, the approach first identifies statements in the module under test that are load sensitive, i.e., they involve the use of *malloc*$(\cdot)$ and *free*$(\cdot)$ statements (in C) and pointers referencing allocated memory. Then, data flow analysis is used to find all Definition-Use (DU)-pairs that trigger the load sensitive statements. Test cases are then built to execute paths for the DU-pairs.

- **Zhang (2002):** (123) consider a multimedia system consisting of a group of servers and clients connected through a network as a SUT (System Under Test). Stringent timing constraints as well as synchronization restrictions are present during the transmission of information from servers to clients and vice versa. The authors identify test cases that can lead to the saturation of one kind of resource, namely CPU usage of a node in the distributed multimedia system. For that, the authors first model the flow and concurrency control of multimedia systems using Petri nets coupled with timing constraints. A specific flavor of temporal logic is used to model temporal constraints. The following are some of the limitations of their technique: (1) the technique cannot be easily generalized to generate test cases to stress test other kinds of resources, such as network traffic, as this would require important changes in the test model; (2) the resource utilization (CPU) of media objects is assumed to be constant over time, although such utilization would likely depend on the requests the server receives for example; (3) no variation of the technique is proposed or even mentioned to stress test over a specific period of time.

- **Grosso (2005):** (109) proposed to combine static analysis and program slicing with evolutionary testing, in order to detect buffer overflow threats. For that purpose, the authors made use of Genetic Algorithms (GA) in order to generate test cases. Actually, static analysis identifies vulnerable statements, while slicing and data dependency analysis identify the relationship between these statements and program or function inputs, thus reducing the search space. To guide the search towards discovering buffer overflow in this work, the authors defined three multi-objective fitness functions and compared them on two open-source systems. These functions account for terms such as the statement coverage, the coverage of vulnerable statements, the distance from buffer boundaries and the coverage of unconstrained nodes of the control flow graph.

- **Briand (2005, 2006):** (124; 125), suggested a methodology also based on GA with the aim to analyse real-time architectures and determine whether deadlines can be missed. The proposed method generates test cases, concentrating on seeding times for aperiodic tasks, such that completion times of a specific task execution are as close as possible to their deadlines. Indeed, they showed that task deadlines may be missed even though the associated tasks have been identified as schedulable through appropriate schedulability analysis. The authors noted that although it is argued that schedulability analysis simulates the worst-case scenario of task executions, this is not always the case because of the assumptions made by schedulability theory. Finally, the authors developed a methodology that helps identifying performance scenarios which can lead to performance failures.

- **Garousi (2006):** (110) presented a stress test methodology that aims at increasing chances of discovering faults related to distributed traffic in distributed systems. The proposed technique uses as input a specified UML 2.0 model of a system, extended with timing information, and grants stress test requirements composed of specific Control Flow Paths along with time values indicating when those paths have to be triggered so as to stress the network to the largest extent possible. In particular, the introduced technique mainly entails (1) a Network Deployment Diagram (following the UML package notation) that describes the distributed architecture in terms of system nodes and networks, and (2) a Modified Interaction Overview Diagram (following the UML 2.0 interaction overview diagram notation) that describes execution constraints between sequence diagrams. Finally using the specification of a real-world distributed system, the authors designed and implemented a prototype system and described how the stress test cases were generated and executed.

- **Bayan (2006):** (126) proposed an approach for the automatic generation of test cases to achieve specified levels of load and stress for a combination of resources. The technique is based on the use of a PID (Proportional, Integral, and Derivative) controller to drive the input and make the system achieve a specified level of resource usage. In fact, PID controller accepts a setpoint as an input which represents the level of resource usage need to be achieved by the application. The PID controller will use an initial test case, defined by the tester, to drive the application to the desired level of usage, independently of the initial test case value. Every time the current usage is fed back to the PID controller, it generates a new gain. In general, positive gain represents an increase in the input and negative gain reflects a decrease in the input.

- **Jiang (2008, 2010):** (111), (112) presented an approach that accesses the execution logs of an application to uncover its dominant behaviour and signals deviations from the application basic behaviour. The intuition behind the proposed approach is that load testing involves the execution of the same operations a large number of times. Therefore, it is expectable that the application under test would generate similar sequences of events a large number of times. These highly repeated sequences of events are the dominant behaviour of the application. Variations from this behaviour are anomalies which should be closely investigated since they are likely to reveal load testing problems. The authors also showed that their solution can automatically identify problems in a load test. However, it requires domain knowledge by the load tester to perform properly.

- **Wang (2010):** (127) described a realistic usage model to simulate user behaviours in load testing of Web applications. Another workload model was also proposed to generate realistic load for load testing. In addition, the paper demonstrates LTAF (Load Testing Automation Framework) tool which is based on the two previous models and can carry out load testing of Web applications automatically. The authors of this article recognize that some enhancement of their proposal should be considered in the future concerning the load model, as they could possibly find no solution or multiple solutions for the model. Furthermore, the authors did not consider the think time, which is an import factor for realistic load testing.

6.5 Discussion

In this section, we present some surveys on testing Web service compositions in general and we discuss later existing solutions on particularly load testing of Web service compositions.

6.5.1 Comparative Evaluation of Existing Approaches

We notice that load testing concerns various fields such as mutlimedia systems (123), network applications (109), embedded systems (126), etc. Furthermore, all these solutions focus on the automatic generation of load test suites. Besides, most of the existing works aim to detect anomalies which are related to resource saturation or to performance issues as throughput, response time, etc.

Only (108) and (128; 129; 130) proposed a solution that allows to verify functional errors in programs/implementations under load conditions. In fact, detected faults according to (108) are related to dynamic memory allocation, and may occur because of memory leaks, incorrect dynamic memory allocation, etc. In the context of BPEL compositions (128; 129; 130), possible errors which may be detected under various load conditions are mainly the addition/omission of non-specified/specified behaviours particularly within BPEL conditional branches. Another potential error type is related to delays coding. Actually, such errors are introduced when implementing synchronous communications conditioned by delays which are different from those specified in the composition model.

In general, the used technique for load test generation is tightly coupled with the SUT model or specification. For example, (123) model their SUT using Petri nets and adopt constraint solving techniques for test generation. Also (110) make use of UML 2.0 as model and on the other hand generate test cases based on identification of the control flow in corresponding sequence diagrams.

The identification of problem cause(s) (application, network or other) is not the main goal behind load testing, rather than studying performance of the application under test, this fact explains why few works address this issue. However, (128; 129) are able to recognize if the detected problem under load is caused by implementation anomalies. Network or other causes are ongoing works. Indeed, the authors are defining and validating an approach based on interception of exchanged messages between the composition under test and its partner services. That way it would be possible to monitor exchanged messages instantaneously, and to know what is the cause behind their loss or probably their reception delay, etc.

Few research efforts, such (111), (112) and (128; 129; 130), are devoted to the automated analysis of load testing results in order to uncover potential problems. Indeed, it is hard to detect problems in a load test due to the large amount of data which must be analysed. Current industrial practice mainly involves time-consuming manual checks which, for instance, search through the logs of the application for error messages.

Finally, we remark that the majority of existing works illustrated their load testing solutions based on academic or/and industrial case studies. Besides, different proposed approaches have been concretized in form of testing tools. However, each developed prototype tool for load testing depends on the SUT domain and is thus dedicated only to a particular context. For example, RTTT tool (124; 125) is used for load testing of reactive real-time systems, whereas WSCLT tool (129) aims to test non-conformances within BPEL compositions considering diverse load conditions.

6.5.2 Testing of Web Service Compositions

A single web service may not be able to satisfy the need of a user. In such a situation, it is possible to combine existing services together in order to fulfill this need. This act is called Web service composition. Although many research works have been focused on the discovery, selection and composition of Web services, research areas such as testing of Web services (especially Web service compositions) are still new and immature (131). In general, testing is the process of executing a program with the intent of finding errors. It involves activities such as specifying test cases, generating test data, monitoring test execution, measuring test coverage, validating test results and tracking system errors. We highlight that several research papers have been written on testing of Web services addressing areas such as testing SOAP messages, WSDL interfaces, and publish, find and bind capabilities. However, Web service composition testing involves testing the extended interaction between the service provider and requester as well as the composition schema which defines the business logic of the composite service.

In this context, existing commercial tools such as Oracle BPEL process manager and Active BPEL Designer support manual functional testing that allows testers to test BPEL descriptions and service compositions based on requirements. Yet existing tools/approaches are not sufficient to reveal structural errors of BPEL processes (132).

Some surveys on Web services testing can be found in (133), (134). Furthermore, (135) and (136) provided surveys focusing on testing of Web service compositions. While Zakaria only concentrated on unit testing of BPEL, Bucchiarone discussed Web service composition testing from the aspect of orchestration and choreography, and classified research papers according to them. However, Bucchiarone's study was done in 2007 and there has been several other research papers published since then. In addition, (137) discussed the importance of Web service compositions testing and provided a classification of the most prominent approaches in this area. For that, the authors presented several criteria for the comparison of these solutions, and conducted a comparative evaluation of the corresponding proposed approaches. The results of the paper gave an essential perspective to do research work on testing of composed Web services. Besides, (138) provided a mapping study of current Web service compositions testing solutions conducted by other researchers. In fact, a mapping study 'involves a search of the literature to determine what sorts of studies addressing the systematic review question have been carried out, where they are published, in what databases they have been indexed, what sorts of outcomes they have assessed, and in which populations' (139). That way, (138) provided an overview of the state of research in the area of Web service composition testing, with the intention of enriching future research area by looking at the gaps and non treated issues yet.

One important type of testing Web service compositions is load/stress testing, as such applications solicit concurrent access by multiple users at the same time. Concerning existing tools in this context, only Oracle has proposed a load testing module. But we noticed, that it generates test reports which are not informative enough. In particular, no knowledge about the application functional aspects is provided under load. Only the chronology and instant of each thread invocation are presented. At the end of test, the rate of threads invocation per second is evaluated. However, (128; 129; 130) proposed a complete solution for significantly load testing Web service compositions. Indeed, the main contribution of their work is the verification of Web service compositions requirements (which are supposed to be formally modeled using Timed Automata) under diverse load conditions. In addition, the authors suggested an automated analysis of load testing logs. Also two different case studies, implemented as BPEL processes, were used to validate and illustrate the solution. Finally, their work was concretized in form of a complete testing tool WSCLT (129) extended with graphical user interfaces.

6.6 Summary

In this chapter, we investigated the opportunities as well as challenges of load and stress testing in general. A classification of existing related works was reported, yet it is not possible to claim that the list is exhaustive. Furthermore, several criteria were introduced in order to evaluate and compare the different approaches. Finally, a summary of both comparison and evaluation of the approaches were presented and the results were also discussed.

At the end, we remarked that there is no consensus on the best approach since each contribution has its strengthen and weakness. Furthermore, each proposed solution is suitable for a specific applicability domain. On the other hand, we noticed that most of the research papers are from conference proceedings, which is an indication that the research area is still immature. More work needs to be done in order to improve the current state of research in both load and stress testing especially for the context of Web service compositions, which constitute actually an emerging paradigm in the domain of Service-Oriented Architecture.

CHAPTER 7

A Model Based Approach to Combine Load and Functional Tests

7.1 Introduction

Many different errors may appear if the application is loaded whereas they may not appear under normal execution conditions. Such errors are qualified as *load sensitive errors* (116). Thus, it is essential to combine both functional and load testing types by adjusting the functional test automation.

In fact, it is insufficient to record only pass/fail verdicts of the tests, but also to supervise times of screens and objects, network communications, etc. That way, the functional test automation suite turns into a performance monitor, which ensures getting thorough test analysis considering various load conditions. In this chapter, we propose a new model-based framework that combines both functional and load tests. Our new framework is based on the model of extended timed automata with inputs/ouputs and shared integer variables.

The remainder of this chapter is organized as follows. In Section 7.2, we describe our proposed formal model-based framework to combine functional and load tests. Our solution is based on the model of extended timed automata with inputs/ouputs and shared integer variables. Some modelling issues are discussed in Section 7.3. In Section 7.4, we report on a case study from the healthcare field. Finally, Section 7.5 provides a conclusion for the chapter.

7.2 Extended Timed Automata

We extend the framework presented in (2). We use timed automata (11) with deadlines to model urgency ((2)). An *extended timed automaton over* Ac is a tuple $A = (Q, q_0, X, \mathsf{I}, \mathsf{Ac}, \mathsf{E})$, where:

- Q is a finite set of *locations*;
- $q_0 \in Q$ is the initial location;
- X is a finite set of *clocks*;
- I is a finite set of integer variables;
- E is a finite set of *edges*.

Each edge is a tuple $(q, q', \psi, r, inc, dec, d, a)$, where: $q, q' \in Q$ are the source and destination locations; ψ is the *guard*, a conjunction of constraints of the form $x \# c$, where $x \in X \cup \mathsf{I}$, c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$; $r \subseteq X \cup \mathsf{I}$ is a set of clocks and integer variables to *reset* to zero; $inc \subseteq \mathsf{I}$ is a set of integer variables (disjoint from r) to *increment* by one; $dec \subseteq \mathsf{I}$ is a set of integer variables (disjoint from r and inc) to *decrement* by one; $d \in \{\mathsf{lazy}, \mathsf{delayable}, \mathsf{eager}\}$ is the *deadline*; $a \in \mathsf{Ac}$ is the action.

An example of an extended timed automaton $A = (Q, q_0, X, \mathsf{I}, \mathsf{Ac}, \mathsf{E})$ over the set of actions $\mathsf{Ac} = \{a, b, c, d\}$ is given in Figure 7.1 where : $Q = \{q_0, q_1, q_2, q_3\}$ is the set of locations; q_0 is the initial location; $X = \{x\}$ is the finite set of clocks; $\mathsf{I} = \{i\}$ is the finite set of integer variables; E is the set of edges drawn in the Figure. The figure uses the following notation: "$x := 0$" means resetting the clock x to 0; "$i := 0$" means resetting the integer variable i to 0; "$i++$" means incrementing i by 1; "$i--$" means decrementing i by 1.

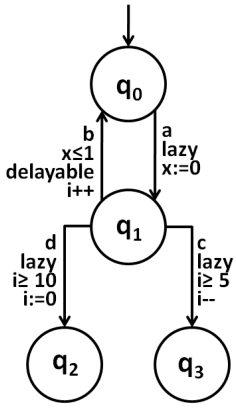


Figure 7.1: An example of an extended timed automaton.

7.3 Modelling Issues

In this section we illustrate some methodological aspects of our framework. First we explain how it is possible to combine both functional and load aspects within the same model. For instance in Figure 7.2, the response time to produce the output action b with respect to the input action a depends on the number of concurrent instances of the considered system under test as follows: output b is generated within at most 1 time unit if the number of concurrent instances is smaller or equal to 100; output b is generated within at most 2 time units if the number of concurrent instances is between 101 and 1000; output b is generated within at most 3 time units if the number of concurrent instances is greater or equal to 1001;

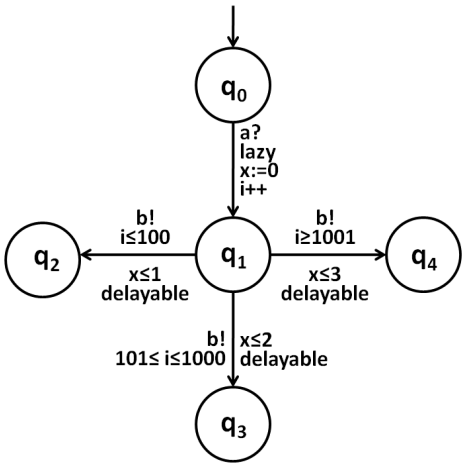


Figure 7.2: An example showing how the time response of the SUT may depend on the number of concurrent instances.

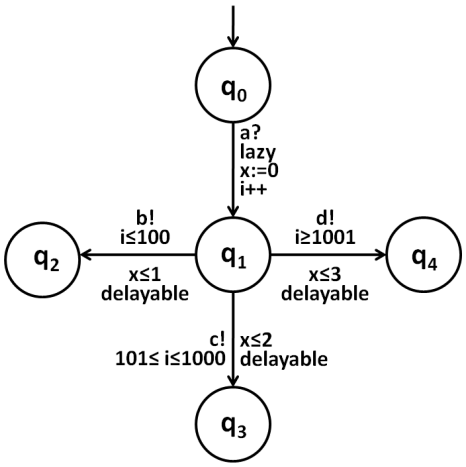


Figure 7.3: An example where the SUT produces different output actions depending on the current load.

In Figure 7.3, we show how to model the fact that the SUT may even produce different output actions with respect to the same input action depending on the current number of concurrent instances of the considered system. The SUT may produce either b, c or d. On the first hand, output a may be seen as the normal output generated by the SUT when the load the smaller or equal to 100. On the other hand, output b may correspond to the situation where the SUT still produces the same desired output action. However this time the output action is mixed with a warning message to inform the user that the system is starting entering a critical area (load between 101 and 1000). Finally, output c may correspond to the production of an error message meaning that the SUT is no longer able to produce the desired output action since the load is too high (greater or equal to 1001).

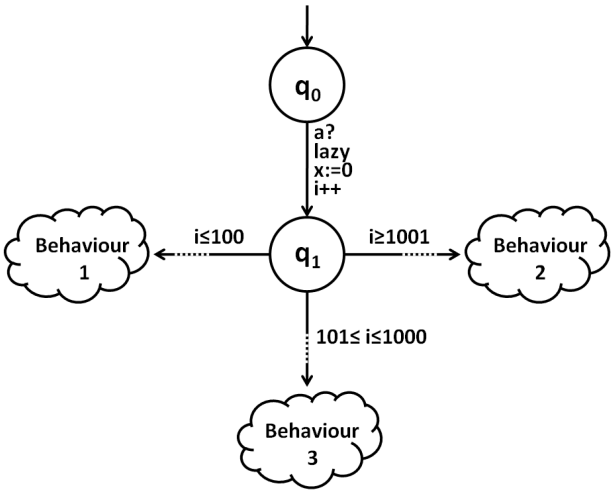


Figure 7.4: An example where the SUT adopts different sophisticated behaviours depending on the current load.

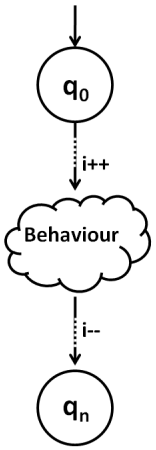


Figure 7.5: The general scheme of the extended timed automaton modelling the system under test.

In Figure 7.4, we consider a more sophisticated situation where the SUT can produce complete different behaviours depending on the current load. Three distinct behaviours are possible according to the figure. Behaviour 1 can be considered as the nominal behaviour of the considered system as in the previous example. The two other behaviours may correspond to the situation where the system under test is trying to find a suitable way to deal with the increase of the current number of concurrent instances and to improve the quality of the service. For instance a possible solution may consist in allocating additional resources to overcome the current critical situation.

In Figure 7.5, we propose a general scheme of the extended timed automaton modelling the system under test. Normally we have to start with an increment of the total number of active instances of the SUT and to finish with a decrement of this counter. Between these two events the behaviour of the SUT is modelled.

Figures 7.6 and 7.7 illustrate different strategies for creating new instances of the SUT. In the first case (Figure 7.6) any instance of the SUT may participate to the generation of new instances. Whereas in the second case (Figure 7.7) a particular central instance is in charge of creating new instances.

In Figures 7.8 and 7.9, we show how it is possible to consider different situations for deleting current instances of the SUT. In the first situation (Figure 7.8) each instance of the SUT is in charge of killing itself. In the second situation (Figure 7.9) a central component is in charge of killing the different instances of the SUT.

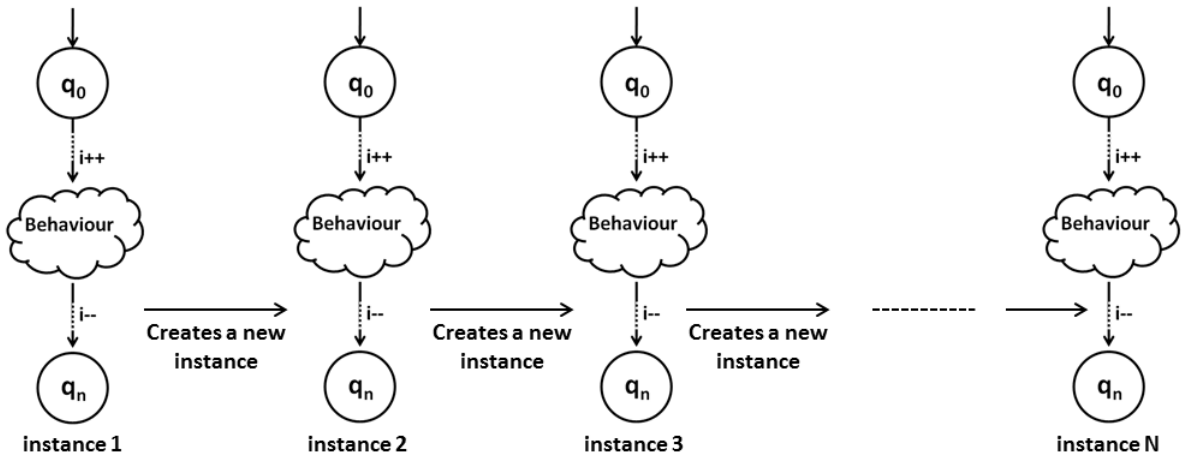


Figure 7.6: Any instance of the SUT may participate to the generation of new instances.

Figure 7.10 explains how counter i is used to follow the increase and the decrease of the current active concurrent instances of the SUT.

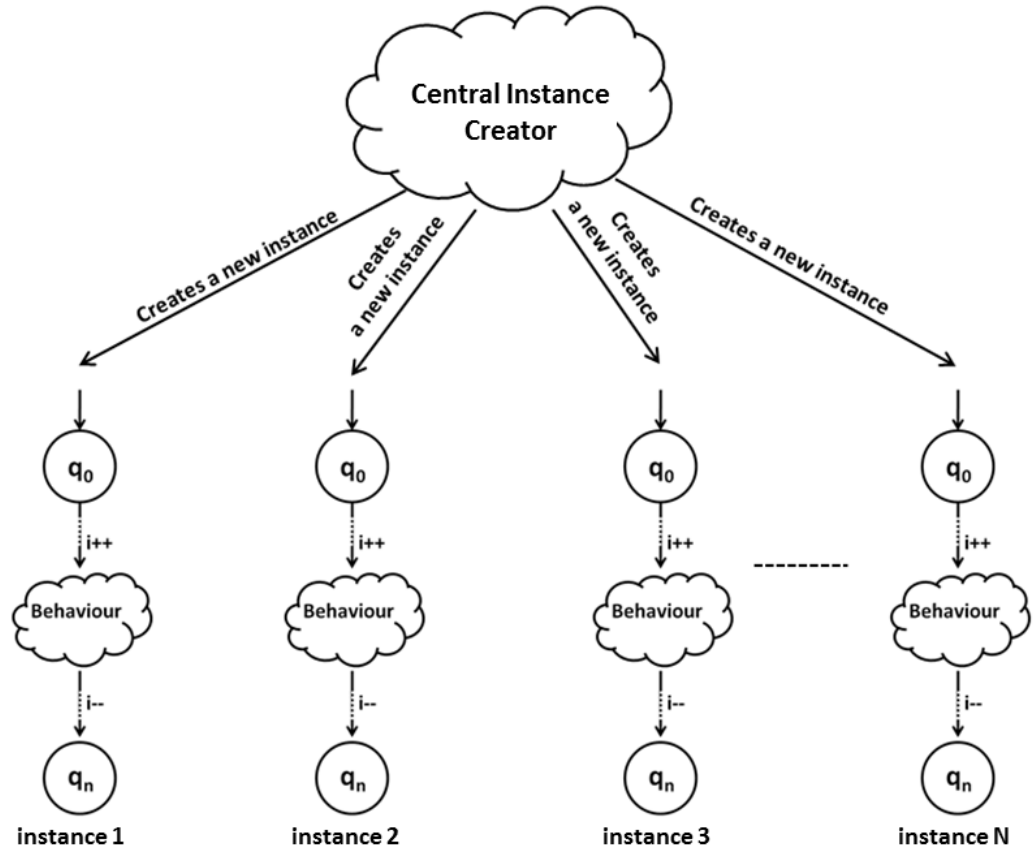


Figure 7.7: A central instance of the SUT is in charge of creating new instances.

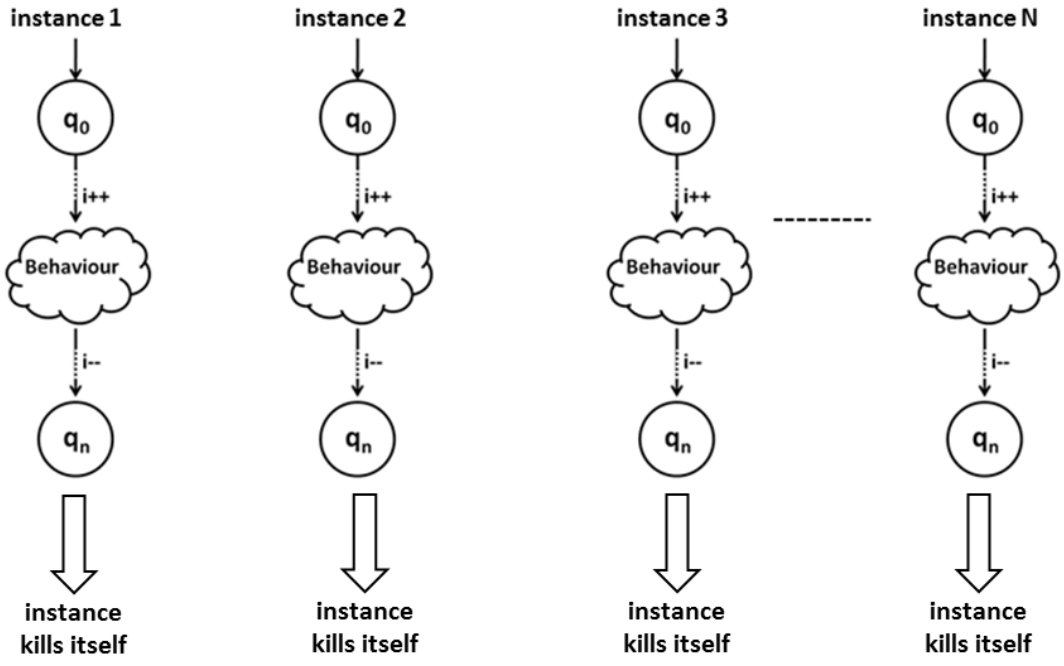


Figure 7.8: Each instance of the SUT is in charge of killing itself.

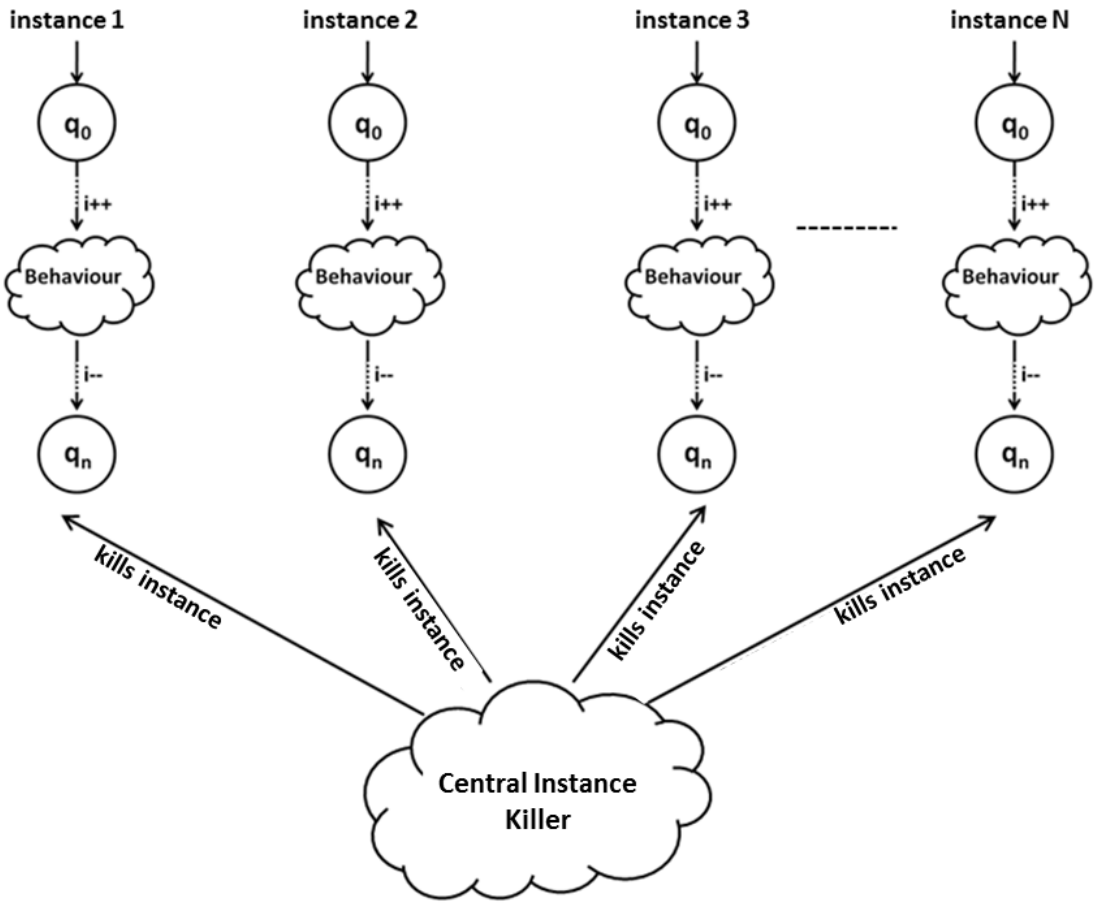


Figure 7.9: A central component is in charge of killing the different instances of the SUT.

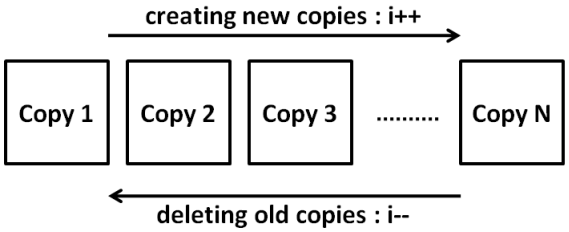


Figure 7.10: The integer variable i allows to follow the increase and the decrease of the number of active instances of the SUT.

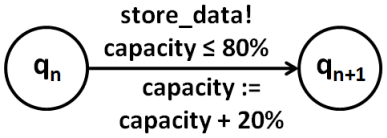


Figure 7.11: The use of other integer variables to model other aspects of the SUT.

Figure 7.11 gives a situation where we can use other integer variables to model other aspects concerning the SUT. In the example proposed by the figure we take into account memory storage capacity. A new shared integer variable is used for this purpose. The action "store_data" may happen only if the current storage capacity is greater or equal to 80%. As soon as this action takes place, the storage capacity is increased by 20%.

7.4 Illustration through the TRMCS case study

7.4.1 Case study description

For a given patient suffering from chronic high blood pressure, measures like his arterial blood pressure and his heart-rate beats per minute are collected periodically (for instance three times per day). For the two collected measures, a request is sent to the TRMCS process. First, the SS is invoked to save periodic reports in the medical database. Then, the AnS is charged with analyzing the monitored data in order to detect whether some thresholds are exceeded. This analysis is conditioned by a waiting/processing time. Indeed, the process should receive a response from the AnS before reaching 30 seconds. Otherwise, the process sends a connection problem report to the MS. In case of receiving the analysis response before reaching 30 seconds, two cases/scenarios are studied. If thresholds are respected/satisfied, a detailed reply is sent to the corresponding patient. Otherwise, the AlS is invoked in order to send emergency request-s/urgent notification to the medical staff (such as doctors, nurses, etc.). Similarly to the AnS, the AlS is constrained by a waiting time. If medical staff are notified before reaching 30 seconds, the final reply is sent to the corresponding patient. Otherwise, the MS is invoked.

7.4.2 Reference specification expressed in Timed Automata

We give the specification of the previously described TRMCS scenario using Uppaal, an integrated tool environment for modeling, validation and verification of systems modeled as networks of Timed Automata (85). In Uppaal, synchronous communication between the Timed Automata is performed by hand-shake synchronization using input and output actions. Output and input actions are denoted with an exclamation mark and a question mark respectively, e.g., a! and a?. Asynchronous communication is achieved by means of shared variables. Throughout the chapter we use Uppaal syntax to illustrate Timed Automata, and Figure 7.12 is direct exported from Uppaal, where x is a local clock. In addition, initial locations are marked using a double circle. Edges are by convention labelled by the triple: guard, action, and assignment in that order. Finally, bold-faced clock conditions placed under locations are location invariants.

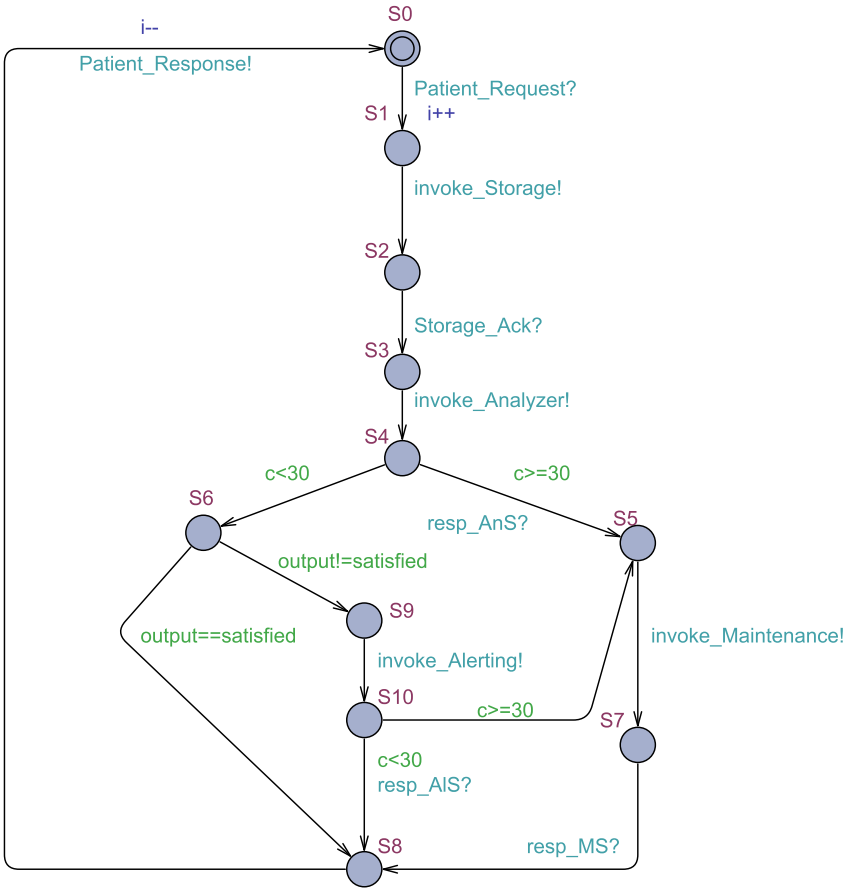


Figure 7.12: The TRMCS process modeled in Timed Automata.

Before referring to the elaborated specification expressed in Timed Automata for testing different TRMCS BPEL implementations, we should be sure that this model respects both functional and non-functional system requirements. For that, Uppaal proposes a simulation module of systems modeled in Timed Automata which enables to follow how the built model can evolve in time. The realized simulations allowed us to detect and correct some errors when modeling our considered TRMCS scenario in Timed Automata. Furthermore, we made use of Uppaal's verification module which enables to check various properties (e.g. safety, liveness, deadlock, etc.) of our created model. That way, we obtain at the end a checked and valid specification expressed in Timed Automata as a reference for testing later.

7.4.3 Illustration of some modelling patterns

In the following, we explain some methodological aspects of our proposed framework. The aim is to show how it is possible to combine both functional and load aspects within the same model. In our case we refer to the TRMCS process modeled in Timed Automata as depicted in Figure 7.12.

As a first pattern example, we show in Figure 7.13 how to model the fact that the TRMCS produces different output actions with respect to the same input action *Patient_Request* depending on the current number of concurrent instances. Indeed, the SUT may produce either *invoke_LocalStorage* or *invoke_CloudStorage*. On the first hand, output *invoke_LocalStorage* may be seen as the normal output generated by the TRMCS when the load is smaller or equal to 50. It corresponds to SS invocation in order to save periodic reports in the medical local database.

On the other hand, output *invoke_CloudStorage* concerns the situation where the load is greater than 50. We suppose that this situation corresponds to SS invocation to store reports in a remote medical database hosted on Cloud. In fact, as the number of concurrent users increases significantly, the local database may be saturated. Thus, Cloud storage seems to be a good solution in this case as it provides users with various capabilities to store and process their data in either privately owned, or third-party data centers that may be located far from the TRMCS user.

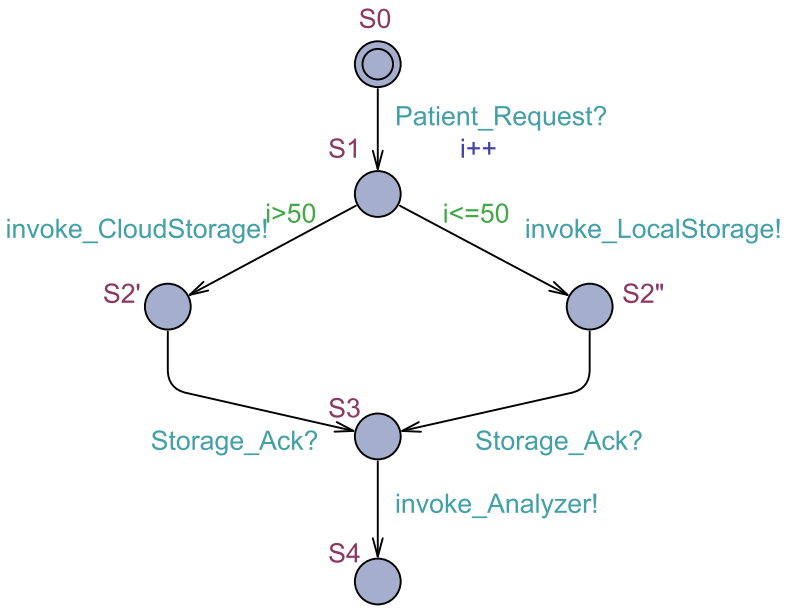


Figure 7.13: Pattern 1: The TRMCS produces different output actions depending on the current load.

Figure 7.14 presents a situation where we can use other integer variables in order to model other aspects concerning the TRMCS. Indeed, we take into account storage capacity. A new shared integer variable is used for this goal. The action *invoke_Storage* may happen only if the current storage capacity is greater or equal to 80%. As soon as this action occurs, the storage capacity is increased by 20%. Definitely, other integer variables may be used to model other aspects such as network connectivity, memory, CPU use, etc.

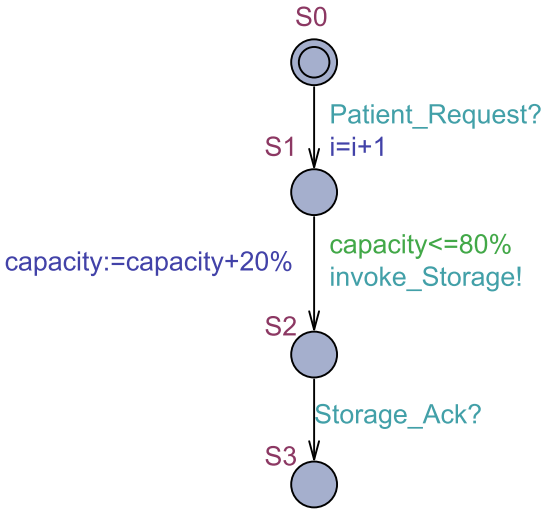


Figure 7.14: Pattern 2: The use of a new shared integer variable to model the storage capacity.

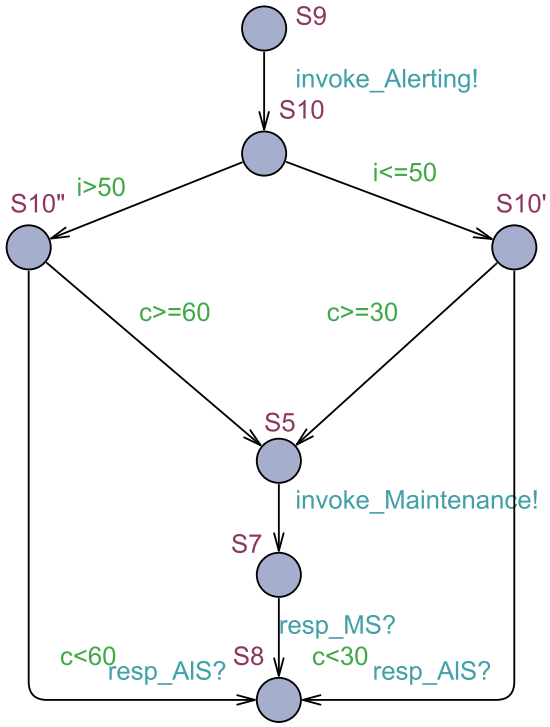


Figure 7.15: Pattern 3: The time response of the TRMCS depends on the number of concurrent instances.

In Figure 7.15, the response time to produce the output action *resp_AlS* with respect to the input action *invoke_Alerting* depends on the number of concurrent instances of the considered TRMCS under test as follows:

- output *resp_AlS* is generated within at most 30 seconds if the number of concurrent instances is smaller or equal to 50. In case of exceeding this deadline, the MS is invoked in order to treat the report concerning the AlS connection problem;

- output *resp_AlS* is generated within at most 60 seconds if the number of concurrent instances is greater than 50. If this deadline is exceeded, then the MS is invoked to treat the report of the AlS connection problem.

Clearly, different other modelling issues could be considered and integrated in the TRMCS Timed Automata depicted in Figure 7.12. This way, we obtain a rich formalism ensuring high expressiveness of the critical system, since it models concurrency and combines both functional and load tests.

7.5 Summary

In this chapter, we proposed a formal model-based framework to combine functional and load tests. Our approach is based on the model of extended timed automata with inputs/ouputs and shared integer variables. The adopted model allows high expressiveness for concurrent systems as it ensures partial-observability and parallel composition. In addition, we presented different modelling issues illustrating some methodological aspects of our framework. Besides, we illustrated our approach by the means of a critical case study from the healthcare field. An important contribution in this work was to use a rich formalism to model mutli-user systems and to combine functional and load tests in the same model. This point constitutes an important testing area that is usually misunderstood.

CHAPTER 8

Limitations of WS-BPEL Compositions under Load Conditions

8.1 Introduction

In this chapter, we propose to realize the monitoring of BPEL compositions behaviors during load testing, in order to perform later an advanced analysis of test results. This step aims to identify both causes and natures of detected problems. For that, we take into consideration the execution context of the application under test while periodically capturing, under load, some performance metrics of the system such as CPU usage, memory usage, etc.

The remainder of this chapter is organized as follows. Section 8.2 is dedicated to describe our proposed testing approach for the study of BPEL compositions under load conditions. Then, we describe in Section 8.3 our automated advanced load test analysis approach and we provide a taxonomy of the different detected problems under load. In Section 8.4 we report on the Travel Agency case study. Finally Section 8.5 concludes the chapter.

8.2 Study of WS-BPEL Compositions under Load

Our proposed approach is based on gray box testing, which is a strategy for software debugging where the tester has limited knowledge of the internal details of the program. Indeed, we simulate in our case the different partner services of the composition under test as we suppose that only the interactions between this latter and its partners are known. Furthermore, we rely on the online testing mode considering the fact that test cases are generated and executed simultaneously (140).

8.2.1 BPEL Concepts

According to OASIS (Organization for the Advancement of Structured Information Standards) (141), a BPEL specification is a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners. It is XML based and it allows sharing distributed data, even through multiple organizations, by employing a combination of Web services.

BPEL syntax consists of a set of activities which can be classified into two categories: basic activities and structured activities. Basic activities allow to invoke an operation of a partner Web service (*invoke* activity), to present the composition like a new Web service with the *receive* activity for describing the reception of a request and the *reply* activity to generate an answer. There are other activities such as *assign*, *wait*, *link*, etc. Structured activities use the basic activities to describe sequential execution (*sequence*) and parallel executions (*flow*), connections (*switch*, *if*), loops (*forEach*, *repeateUntil*, *while*), and finally alternate ways (*pick*).

8.2.2 Principle of Load Distribution

Figure 8.1 shows the architecture to set up in order to realize remote load test distribution. Indeed, the role of the test manager is to monitor the test execution and distribute the required load between the different load generators. These latters invoke concurrently the system under test as imposed by the test manager.

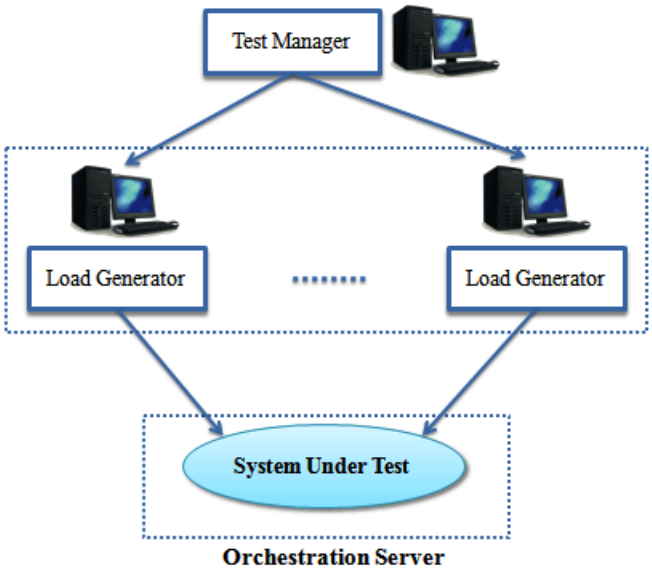


Figure 8.1: Load Distribution Architecture.

8.2.3 WSCLim Architecture

In this section, we describe our proposed distributed framework for limitations study of BPEL compositions under load conditions. To illustrate our solution, we consider, for simplicity reasons, that our testing architecture is spread over four machines (hosts), as depicted in Figure 8.2, such that the first one (Host1) is dedicated to deploy the system under test. In the second machine (Host2), our WSCLim tool is installed. Finally, the two other machines (Host3 and Host4) ensure load distribution: each one acts as a load generator.

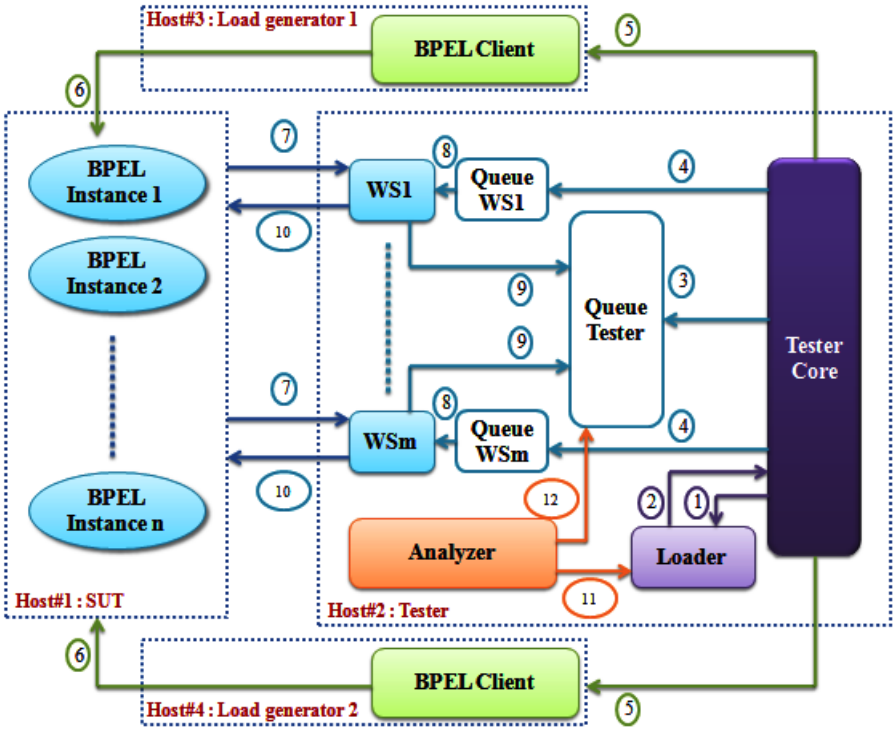


Figure 8.2: Load Testing Architecture.

As shown in Figure 8.2, the main components of our proposed architecture are:

- **System Under Test (SUT):** A new BPEL instance is created for each call of the composition under test. A BPEL instance is defined by a unique identifier. Each created instance invokes its own partner services instances by communicating while exchanging messages.
- **Tester:** It represents the system under test environment and consists of:
  - *Web services (WS1, ..., WSm):* These services correspond to simulated partners of the composition under test.
  - *Queues*: These entities are simple text files through which partner services and the Tester Core exchange messages.

- *Loader*: It loads the SUT specification described in Timed Automata, besides the WSDL files of the composition under test and the WSDL files of each partner service.
- *Tester Core:* It generates random input messages of the BPEL process under test. It communicates with the different partner services of the composition by sending them the types of input and output messages.
- *QueueTester*: It stores the general information of the test (number of calls of the composition under test, the delay between the invocation of BPEL instances, etc.).
- *Analyzer*: This component is responsible for offline analysis of the test log QueueTester.

- **BPEL Clients**: These entities meet the order of the Tester Core by performing concurrent invocations of the composed service. For that, they receive as test parameters the input(s) of the composition under test, the number of required process calls and the delay between each two successive invocations.

8.2.4 Testing Procedure

As illustrated in Figure 8.2, we describe in the following the necessary steps to test BPEL compositions considering load distribution concept. So once the tester provides the necessary information for the test (the specification described in Timed Automata, the WSDL description of the composite service, the number of concurrent calls of the system under test and the delay between each two successive invocations), and starts the test then:

a. The Tester Core calls the Loader to load Timed Automata, the WSDL file of the composed service under test and the WSDL files of the different partner services.

b. From these files, the Loader determines the types of input/output variables of the composite service as well as those of partner services. It also defines synchronous communications. In addition, it sends this information to the Tester Core.

c. After receiving these data by the Loader, the Tester Core sets information of the test in QueueTester.

d. The tester sends for each partner service the corresponding information about the types of input/output messages. In case of synchronous communications, it sends the maximum tolerated time, corresponding to answering the composition under test, to the partner service which is involved in this communication.

e. The Tester Core divides the simulated load between both BPEL Client entities in our case, and calls each one by communicating the number of BPEL process concurrent calls (threadsNumber) and the delay between each two successive invocations (delay).

f. Each BPEL Client invokes the composite service (threadsNumber) times each (delay).

g. The different BPEL instances (BPEL1, BPEL2,..., BPELn) are executed simultaneously. Each instance of the composite service invokes its own instances of partner services. At each call of one of these services, it records a trace in logs indicating the identifier of the BPEL instance that has invoked it, the time of its invocation and its input parameters.

h. Each instance of the partner service determines from its own queue, the types of its i/o variables and checks the types of messages that it received from the BPEL composition.

i. Each instance of the invoked partner service sets in QueueTester, the ID of the BPEL process instance which is responsible of its invocation, the information that it received and the result of input message types checking.

j. Referring to the type of the output variable in its queue, the partner service instance generates randomly a response and sends it to the SUT.

k. After running the test, the analyzer consults the Loader in order to obtain a list of path(s) that the BPEL process may cross (according to the specification).

l. At the end of test, the Analyzer examines the stored information in QueueTester in order to generate a final report containing test verdicts relatively to each invoked BPEL instance.

8.3 Automated Advanced Load Test Analysis Approach

In common current industrial practices, looking for functional problems in a load testing is a time-consuming and difficult task, due to the challenges such as no documented system behavior, monitoring overhead, time pressure and large volume of data. The ad-hoc logging mechanism is the most commonly used, as developers insert output statements into the source code for debugging purposes (142). Most practitioners look for the functional problems under load using manual searches for specific keywords like *failure*, or *error* (112). After that, load testing practitioners analyze the context of the matched log lines to determine whether they indicate functional problems or not. Depending on the length of a load test and the volume of generated data, load testing practitioners may spend several hours to perform these checks.

8.3.1 Principle of Load Test Analysis Approach

Performed operations during load testing of BPEL compositions are stored in QueueTester. In order to recognize each BPEL instance which is responsible for a given action, each one starts with the identifier of its corresponding BPEL instance (BPEL-ID). At the end of test running, the Analyzer consults QueueTester and goes through three steps:

- **Decomposition of QueueTester:** Based on BPEL-ID, the Analyzer decomposes information into atomic test reports. Each report is named BPEL-ID and contains information about the instance which identifier is BPEL-ID.
- **Analysis of atomic logs:** The Analyzer consults the generated atomic test reports of the different BPEL instances. It verifies the observed executed actions of each instance by referring to the specified requirements in Timed Automata. Finally, the Analyzer assigns corresponding verdicts to each instance and identifies detected problems.
- **Generation of final test report:** This last step consists in producing a final test report recapitulating test results relatively to all instances and also describing both nature and cause of each observed FAIL verdict.

8.3.2 Classification of Detected Problems under Load

In this section, we present and classify the most observed problems by experiments during load test executions of different BPEL compositions. Particularly, three sources (causes) of problems are discussed.

8.3.2.1 Functional Problems (SUT)

Load sensitive faults in programs may have no damaging effect under small loads or short executions, but cause a program to fail when it is executed under a heavy load or over a long period of time, which results in non-compliance with the specification. In our context, we essentially consider two possible load sensitive faults in the SUT implementation:

- **Non specified behaviors:** This error means that a non specified behavior is added (resp. omitted) by fault within a branch in the BPEL flow, which results in the occurrence of non required treatments (resp. the absence of expected features). This type of problem may occur in a conditional branch (i.e. either temporal constraint implemented with the *pick* activity or logical condition defined by the *switch* activity). In fact, load can influence the variation of time response of a partner service. Also it may affect the choice decision controlling so the BPEL flow execution.

- **Erroneous delays:** This error may appear within a *pick* activity of the BPEL flow under test. It consists of an implementation of a synchronous communication conditioned by a timeout response of a partner service which is different from the specified one.

8.3.2.2 Test Environment

In addition to the previously introduced errors, we consider in our study the execution context of the composition under test and we aim so to identify non-functional problems. Indeed, partner services, the application servers deploying them and the nodes of the test architecture may influence the load test execution and probably cause errors. In this context, we distinguish two types of errors:

- **Problem of connection to a partner service:** The BPEL process can not invoke a partner service which usually stops its execution. In fact, the availability of a Web service may be influenced by the load, by the state of the server on which it is deployed, etc.
- **Problem of getting a response from a partner service:** The BPEL process does not receive a response from an invoked partner service for a period of time. To clarify, in order to avoid a long waiting time of a response from a partner, this time is limited by a maximum network delay (*tmax*). Thus, the composed service should wait to a maximum of *tmax* seconds to receive a response from any partner service.

8.3.2.3 SUT Node

As previously demonstrated, both the application and the test environment may be sources of different problems during load testing. In other situations, we could just predict the node (either the tester machine or the SUT machine) that causes the problem. Based on different experiments, we observed in some test scenarios a delay in treatment of a partner service response, at the level of the SUT node. Indeed, it is about a partner service which sends a response to the composed service under test, within a synchronous communication, before the specified maximum delay, except that this composition follows the *onAlarm* branch. This situation may be explained by the fact that the running BPEL instances in parallel share the node resources such as the processor,the memory, etc., which leads sometimes to a delay in the treatment of some instances.

8.4 Travel Agency Case Study

In order to validate our proposed testing architecture, we developed a tool (WSCLim) for load testing and limitations detection of Web services compositions. In the following, we are going to introduce our prototype tool and illustrate our solution through a case study.

8.4.1 Graphical User Interface

We present in this section a brief description of the main interface of our proposed WSCLim tool. As shown in Figure 8.3, this interface allows the user to specify:

- The path of the specification (Timed Automata) used as a reference.
- The path of the composition WSDL specification.
- The number of BPEL concurrent instances.
- The delay between each two successive invocations of the BPEL process under test.

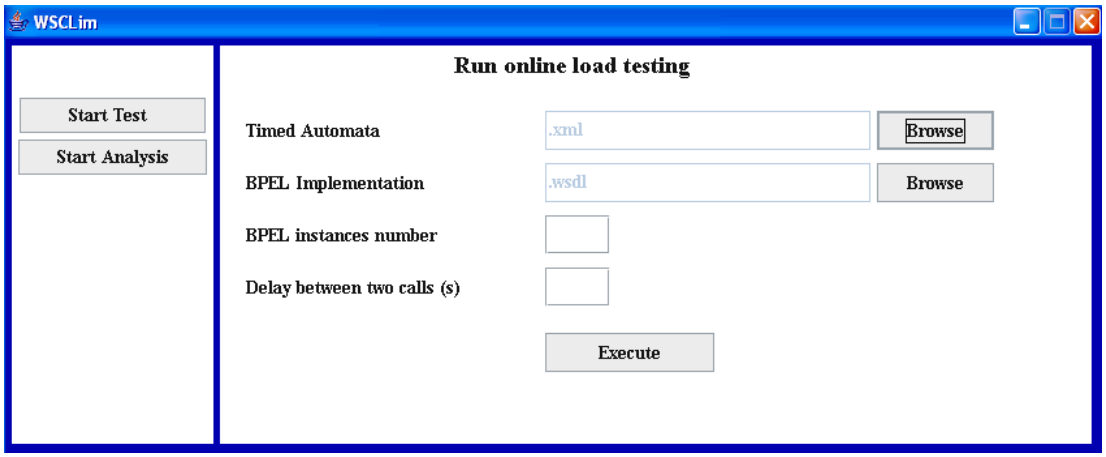


Figure 8.3: WSCLim Tool Initial Interface.

By clicking the button *Execute*, the test is running. During execution, details of the test are stored in log files. At the end of test, the analysis of results is launched by clicking the button *Start Analysis* and the interface containing test verdicts is displayed. We will expose later an example of this interface.

8.4.2 Case Study Description

In this section, we introduce a Travel Agency case study for best illustration of our solution. In fact, we firstly suppose that the required business process (written in BPEL) composes services of: flight search (FS), hotel search (HS), flight booking (FB) and hotel booking (HB).

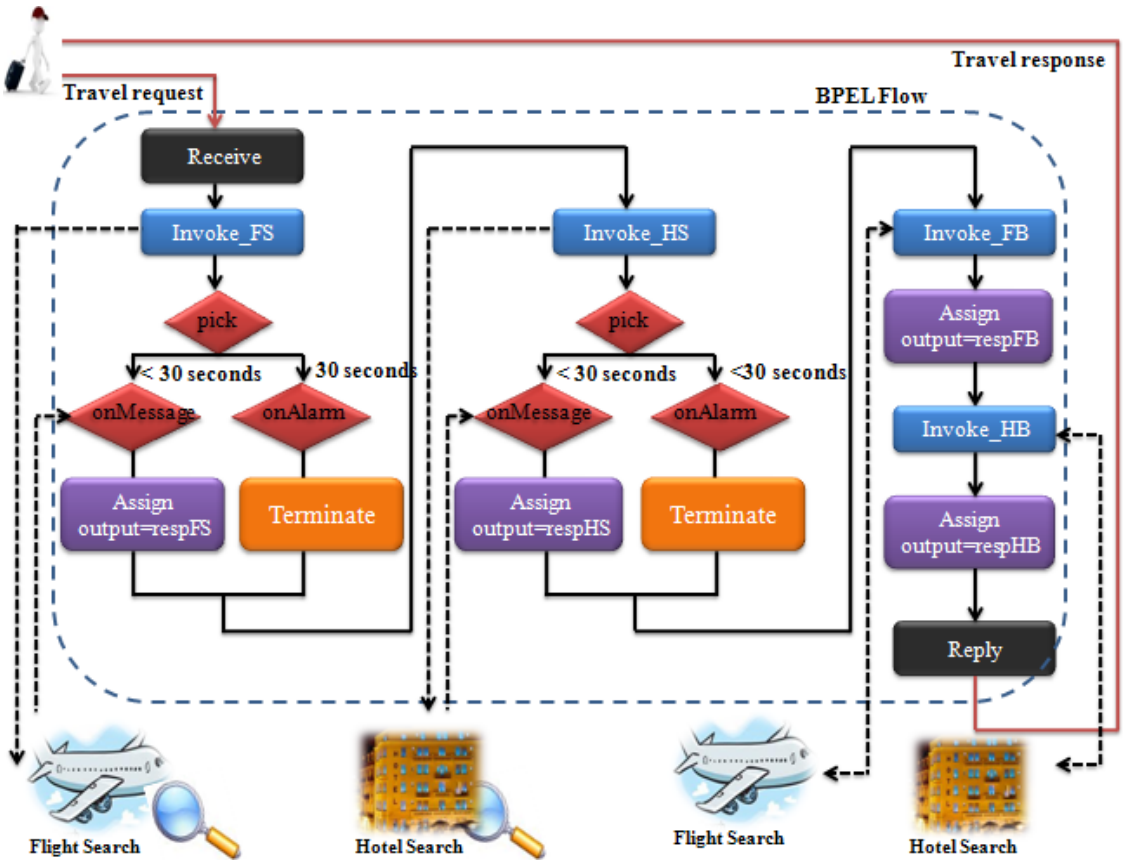


Figure 8.4: The Travel Agency Process.

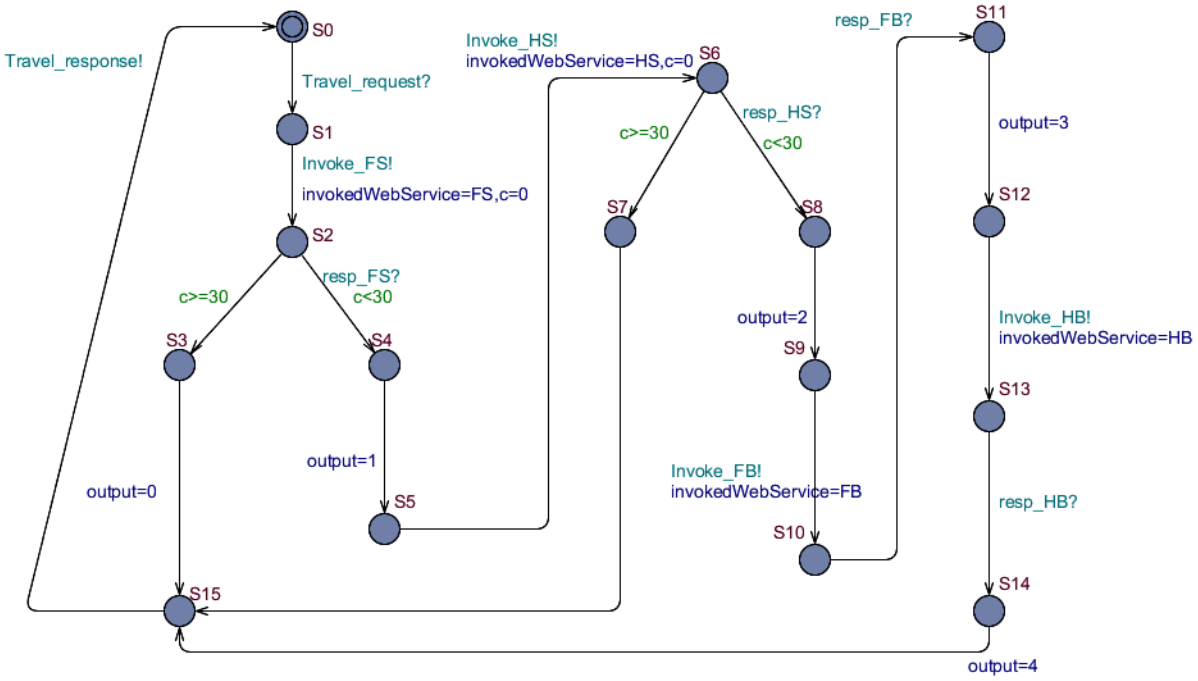


Figure 8.5: The Travel Agency Process modeled in Timed Automata.

As described in Figure 8.4, when a client sends a trip request to the travel agency, the travel search process interacts with information systems of airline companies (resp. hotel chains) for flights (resp. hotel rooms) that match client needs. These two searches are conditioned by a waiting time. Indeed, the process should receive a response from FS (resp. from HS) within maximum 30 seconds. Otherwise, the process execution is stopped. In case of receiving both responses in time, FB and HB services are invoked successively to perform travel booking. Finally, a detailed reply informing about the final results is sent to the concerned client.

Before launching load test, one should provide a written specification in Timed Automata. For that, we first modeled the previously described Travel Agency scenario using Uppaal (see Figure 8.5). Furthermore, we make use of Uppaal to simulate our created Timed Automata which enables to follow how the built model evolves in time, to detect and to correct so some faults when modeling our considered Travel Agency scenario.

8.4.3 Test Scenario

In order to study the limitations of the Travel Agency process, we defined several test scenarios. In this section, we present one of these scenarios. We consider a mutated version of the Travel Agency process where we suppose that a developer made mistakes while coding the BPEL composition as shown in Figure 8.6 (red color). In fact, the service FB was added in the BPEL implementation when exceeding the time limit for the flight search (FS). Moreover, the implemented timeout (60 seconds) of service HS response is different from the specified one (30 seconds) in the Timed Automata (see Figure 8.5). In this scenario, we invoked 40 times the Travel Agency process considering a delay of one second between each two successive invocations.

Figure 8.7 shows the generated analysis interface according to the first test scenario. Indeed, it consists of four blocks:

a. *Test Verdicts* block: this block shows the generated test verdicts in percentage. We note that in this scenario, the percentage of FAIL is equal to 7.5%. This means that 3 BPEL instances among 40 ones failed during load testing.

b. *FAIL Natures & Causes* block: this block presents the nature and the cause (each cause is distinguished by a color) of each observed FAIL verdict.

c. *BPEL Instance vs Response Time* block: this third block informs about response times of the different invoked BPEL instances.

d. *Performance Monitoring* block: the final block graphically shows the performance data recorded during the test by the PerfMon tool.

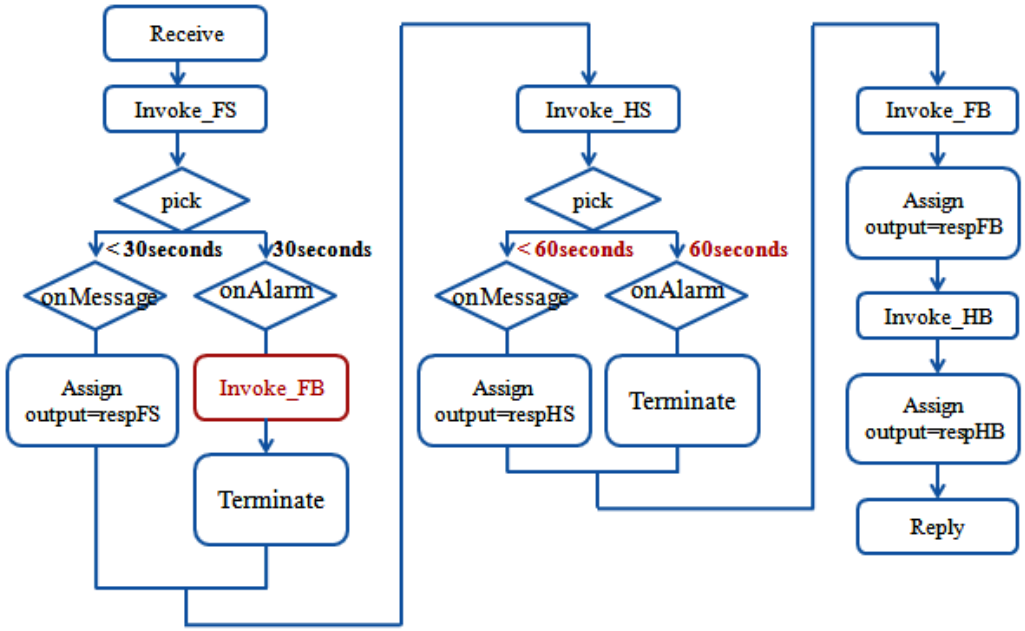


Figure 8.6: Non-compliant BPEL Implementation.

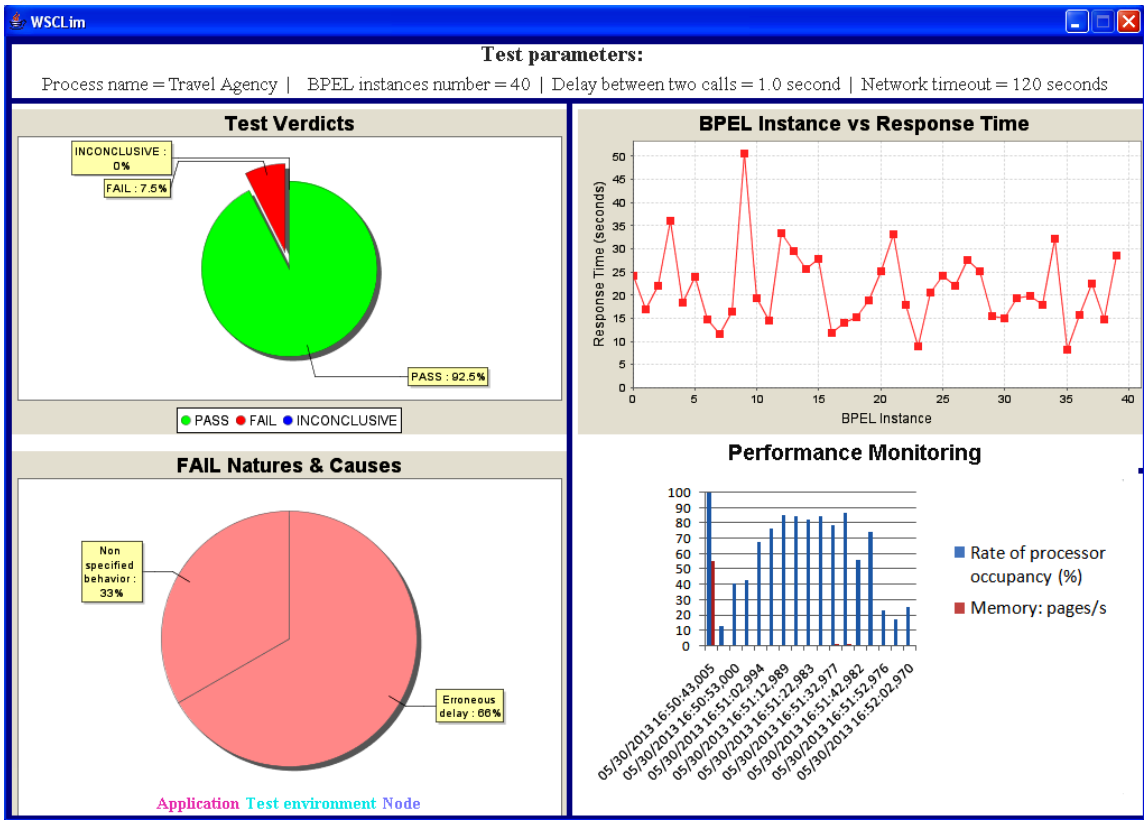


Figure 8.7: Analysis Interface corresponding to the proposed Test Scenario.

8.4.4 Overhead of WSCLim Tool

In order to determine the overhead of our WSCLim tool, we represented, for both cases, the measurement curves of the execution time average while varying the load conditions. In the first case, tests are performed using our testing tool. In the second case, test executions are performed directly from the console of the orchestration server and without turning to our WSCLim tool. To lead these experiments, we considered again the same Travel Agency process structure as described in Section 4.2. As shown in Figure 8.8, the use of our proposed WSCLim tool does not cause a significant additional overhead to the average of the process execution time. Indeed, for a given load, the difference between the two corresponding times is of the order of a few seconds (4 seconds on average). This negligible overhead (compared to the average of one instance execution time) is due to additional activities (i.e. verification of variable types, logging activity, etc.) carried out by our tool during the load testing.

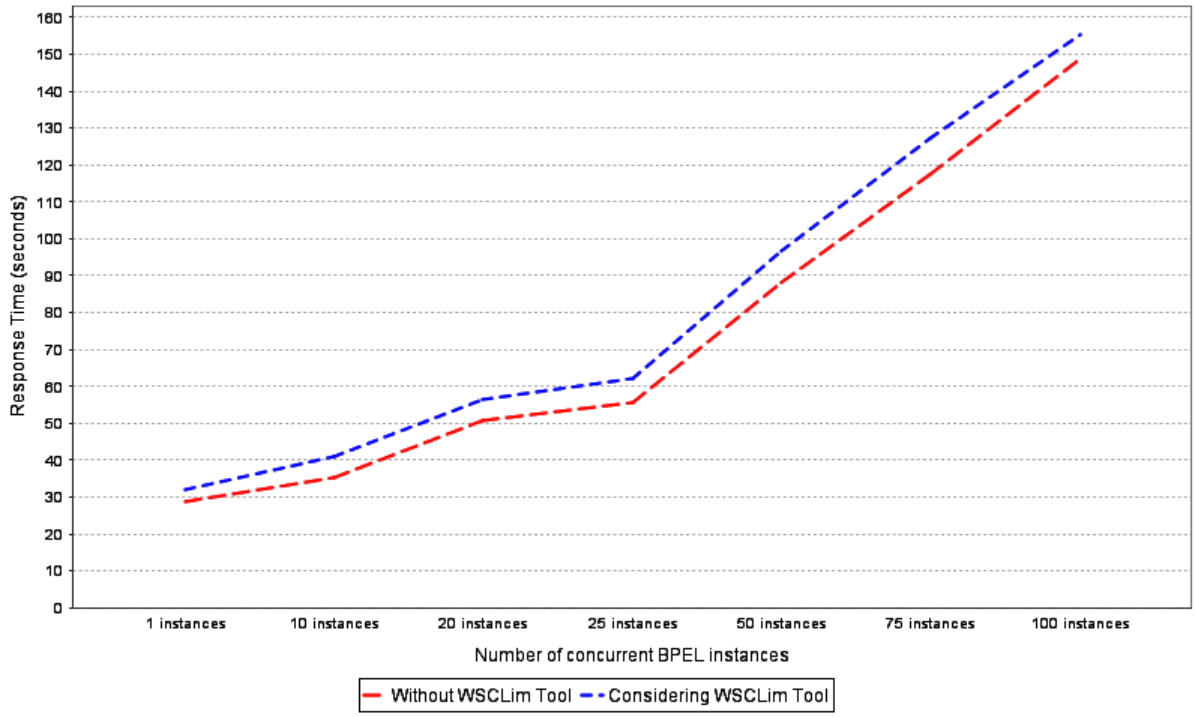


Figure 8.8: Evolution of the Response Time with and without considering the WSCLim Tool.

8.5 Summary

In this chapter we firstly described our contribution for the study of BPEL compositions behaviors under various load conditions. Then, we explained the principle of test logs analysis phase. We also proposed a taxonomy of the detected problems by our solution and we illustrated how test verdicts are assigned. The last phase of our work was dedicated to validate our approach

based on a Travel Agency case study. In fact, we created, simulated and verified the reference model (corresponding to our case study and written in Timed Automata) using the Uppaal test environment. After that, we implemented different mutated versions of the considered BPEL process, and we used our WSCLim tool to automatically execute the corresponding load tests of these implementations. Finally, test results were exhaustively analyzed and advanced information was provided by our tool, which permits to detect and illustrate different natures and causes of errors among those proposed in our previous classification of problems.

Part III

Determinization and Off-Line Test Selection for Timed Automata

CHAPTER 9

A Game Approach to Determinize Timed Automata

9.1 Introduction

Timed automata are frequently used to model real-time systems. Their determinization is a key issue for several validation problems. However, not all timed automata can be determinized, and determinizability itself is undecidable. In this chapter, we propose a game-based algorithm which, given a timed automaton with ε-transitions and invariants, tries to produce a language-equivalent deterministic timed automaton, otherwise a deterministic over-approximation. Our method subsumes two recent contributions: it is at once more general than the determinization procedure of (3) and more precise than the approximation algorithm of (2).

The structure of this chapter is as follows. Section 9.2 presents the motivation behind the work presented int his chapter. In Section 9.3 we recall definitions and properties relative to timed automata, and present the two recent pieces of work to determinize timed automata or provide a deterministic over-approximation. Section 9.4 is devoted to the presentation of our game approach and its properties. Extensions of the method to timed automata with invariants and ε-transitions are then presented in Section 9.5. A comparison with existing methods is detailed in Section 9.6. Section 9.7 concludes the chapter.

9.2 Motivation

Timed automata (TA), introduced in (9), form a usual model for the specification of real-time embedded systems. Essentially TAs are an extension of automata with guards and resets of

continuous clocks. They are extensively used in the context of many validation problems such as verification, control synthesis or model-based testing. One of the reasons for this popularity is that, despite the fact that they represent infinite state systems, their reachability is decidable, thanks to the construction of the region graph abstraction.

Determinization is a key issue for several problems such as implementability, diagnosis or test generation, where the underlying analyses depend on the observable behavior. In the context of timed automata, determinization is problematic for two reasons. First, determinizable timed automata form a strict subclass of timed automata (9). Second, the problem of the determinizability of a timed automaton, (i.e. does there exist a deterministic TA with the same language as a given non-deterministic one?) is undecidable (143; 144). Therefore, in order to determinize timed automata, two alternatives have been investigated: either restricting to determinizable classes or choosing to ensure termination for all TAs by allowing over-approximations, i.e. deterministic TAs accepting more timed words. For the first approach, several classes of determinizable TAs have been identified, such as strongly non-Zeno TAs (145), event-clock TAs (146), or TAs with integer resets (147). In a recent paper, Baier, Bertrand, Bouyer and Brihaye (3) propose a procedure which does not terminate in general, but allows one to determinize TAs in a class covering all the aforementioned determinizable classes. It is based on an unfolding of the TA into a tree, which introduces a new clock at each step, representing original clocks by a mapping; a symbolic determinization using the region abstraction; a folding up by the removal of redundant clocks. To our knowledge, the second approach has only been investigated by Krichen and Tripakis (2). They propose an algorithm that produces a deterministic over-approximation based on a simulation of the TA by a deterministic TA with fixed resources (number of clocks and maximal constant). Its locations code (over-approximate) estimates of possible states of the original TA, and it uses a fixed policy governed by a finite automaton for resetting clocks.

Our method combines techniques from (3) and (2) and improves those two approaches, despite their notable differences. Moreover, it deals with both invariants and ε-transitions, but for clarity we present these treatments as extensions. Our method is also inspired by a game approach to decide the diagnosability of TAs with fixed resources presented by Bouyer, Chevalier and D'Souza in (148). Similarly to (2), the resulting deterministic TA is given fixed resources (number of clocks and maximal constant) in order to simulate the original TA by a coding of relations between new clocks and original ones. The core principle is the construction of a finite turn-based safety game between two players, Spoiler and Determinizator, where Spoiler chooses an action and the region of its occurrence, while Determinizator chooses which

clocks to reset. Our main result states that if Determinizator has a winning strategy, then it yields a deterministic timed automaton accepting exactly the same timed language as the initial automaton, otherwise it produces a deterministic over-approximation. Our approach is more general than the procedure of (3), thus allowing one to enlarge the set of timed automata that can be automatically determinized, thanks to an increased expressive power in the coding of relations between new and original clocks, and robustness to some language inclusions. Contrary to (3) our techniques apply to a larger class of timed automata: TAs with ε-transitions and invariants. It is also more precise than the algorithm of (2) in several respects: an adaptative and timed resetting policy, governed by a strategy, compared to a fixed untimed one and a more precise update of the relations between clocks, even for a fixed policy, allow our method to be exact on a larger class of TAs. The model used in (2) includes silent transitions, and edges are labeled with urgency status (eager, delayable, or lazy), but urgency is not preserved by their over-approximation algorithm. These observations illustrate the benefits of our game-based approach compared to existing work.

9.3 Preliminaries

In this section, we start by introducing the model of timed automata, and then review two approaches for their determinization.

9.3.1 Timed Automata

We start by introducing notations and useful definitions concerning timed automata (9).

Given a finite set of clocks X, a clock *valuation* is a mapping $v : X \to \mathbb{R}_{\geq 0}$. We note $\overline{0}$ the valuation that assigns 0 to all clocks. If v is a valuation over X and $t \in \mathbb{R}_{\geq 0}$, then $v + t$ denotes the valuation which assigns to every clock $x \in X$ the value $v(x) + t$, and $\overleftrightarrow{v} = \{v + t \mid t \in \mathbb{R}\}$ denotes past and future timed extensions of v. For $X' \subseteq X$ we write $v_{[X' \leftarrow 0]}$ for the valuation equal to v on $X \setminus X'$ and to $\overline{0}$ on X', and $v_{|X'}$ for the valuation v restricted to X'.

Given a non-negative integer M, an *M-bounded guard*, or simply guard when M is clear from context, over X is a finite conjunction of constraints of the form $x \sim c$ where $x \in X$, $c \in [0, M] \cap \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$. We denote by $G_M(X)$ the set of M-bounded guards over X. Given a guard g and a valuation v, we write $v \models g$ if v satisfies g. Invariants are restricted cases of guards: given $M \in \mathbb{N}$, an *M-bounded invariant* over X is a finite conjunction of constraints of the form $x \lhd c$ where $x \in X$, $c \in [0, M] \cap \mathbb{N}$ and $\lhd \in \{<, \leq\}$. We denote by $I_M(X)$ the set of invariants. Given two finite sets of clocks X and Y, a *relation* between clocks of X and those of Y is a finite conjunction C of atomic constraints of the form $x - y \sim c$ where

$x \in X,\ y \in Y,\ \sim \in \{<, =, >\}$ and $c \in \mathbb{N}$. When, moreover, the constant c is constrained to belong to $[-M', M]$, for some constants $M, M' \in \mathbb{N}$, we denote by $\mathsf{Rel}_{M,M'}(X, Y)$ the set of relations between X and Y.

Definition 1 *A timed automaton (TA) is a tuple* $\mathcal{A} = (L, \ell_0, F, \Sigma, X, M, E, \mathsf{Inv})$ *such that:* L *is a finite set of locations,* $\ell_0 \in L$ *is the initial location,* $F \subseteq L$ *is the set of final locations,* Σ *is a finite alphabet,* X *is a finite set of clocks,* $M \in \mathbb{N}$, $E \subseteq L \times G_M(X) \times (\Sigma \cup \{\varepsilon\}) \times 2^X \times L$ *is a finite set of edges, and* $\mathsf{Inv} : L \to I_M(X)$ *is the invariant function.*

The constant M is called the maximal constant of $\mathcal{A}$, and we will refer to $(|\ X\ |, M)$ as the *resources* of $\mathcal{A}$. The semantics of a timed automaton $\mathcal{A}$ is given as a timed transition system $\mathcal{T}_{\mathcal{A}} = (S, s_0, S_F, (\mathbb{R}_{\geq 0} \times (\Sigma \cup \{\varepsilon\})), \to)$ where $S = L \times \mathbb{R}^X_{\geq 0}$ is the set of states, $s_0 = (\ell_0, \overline{0})$ the initial state, $S_F = F \times \mathbb{R}^X_{\geq 0}$ the final states, and $\to \subseteq S \times (\mathbb{R}_{\geq 0} \times (\Sigma \cup \{\varepsilon\})) \times S$ the transition relation composed of moves of the form $(\ell, v) \xrightarrow{\tau, a} (\ell', v')$ whenever there exists an edge $(\ell, g, a, X', \ell') \in E$ such that $v + \tau \models g \wedge \mathsf{Inv}(\ell)$, $v' = (v + \tau)_{[X' \leftarrow 0]}$ and $v' \models \mathsf{Inv}(\ell')$.

A *run* ρ of $\mathcal{A}$ is a finite sequence of moves starting in s_0, *i.e.*, $\rho = s_0 \xrightarrow{\tau_1, a_1} s_1 \cdots \xrightarrow{\tau_k, a_k} s_k$. Run ρ is said accepting if it ends in $s_k \in S_F$. A *timed word* over Σ is an element $(t_i, a_i)_{i \leq n}$ of $(\mathbb{R}_{\geq 0} \times \Sigma)^*$ such that $(t_i)_{i \leq n}$ is nondecreasing. The timed word associated with ρ is $w = (t_{i_1}, a_{i_1}) \ldots (t_{i_m}, a_{i_m})$ where $(a_i \in \Sigma \text{ iff } \exists\, n,\ a_i = a_{i_n})$ and $t_i = \sum_{j=1}^{i} \tau_j$. We write $\mathcal{L}(\mathcal{A})$ for the language of $\mathcal{A}$, that is the set of timed words w such that there exists an accepting run which reads w. We say that two timed automata $\mathcal{A}$ and B are *equivalent* whenever $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathsf{B})$.

A *deterministic* timed automaton (abbreviated DTA) $\mathcal{A}$ is a TA such that for every timed word w, there is at most one run in $\mathcal{A}$ reading w. $\mathcal{A}$ is *determinizable* if there exists a deterministic timed automaton B with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathsf{B})$. It is well-known that some timed automata are not determinizable (9); moreover, the determinizability of timed automata is an undecidable problem, even with fixed resources (144; 143).

An example of a timed automaton is depicted in Figure 9.1. This nondeterministic timed automaton has ℓ_0 as initial location (denoted by a pending incoming arrow), ℓ_3 as final location (denoted by a pending outgoing arrow) and accepts the following language: $\mathcal{L}(\mathcal{A}) = \{(t_1, a) \cdots (t_n, a)(t_{n+1}, b) \mid t_{n+1} < 1\}$.

The region abstraction forms a partition of valuations over a given set of clocks. It allows one to make abstractions in order to decide properties like the reachability of a location. We let X be a finite set of clocks, and $M \in \mathbb{N}$. We write $\lfloor t \rfloor$ and $\{t\}$ for the integer part and the fractional part of a real t, respectively. The equivalence relation $\equiv_{X,M}$ over valuations over X is defined as follows: $v \equiv_{X,M} v'$ if (i) for every clock $x \in X$, $v(x) \leq M$ iff $v'(x) \leq M$; (ii) for every clock $x \in X$, if $v(x) \leq M$, then $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ and $\{v(x)\} = 0$ iff $\{v'(x)\} = 0$ and (iii)

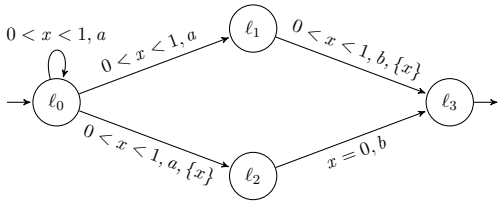


Figure 9.1: A timed automaton $\mathcal{A}$.

for every pair of clocks $(x, y) \in X^2$ such that $v(x) \leq M$ and $v(y) \leq M$, $\{v(x)\} \leq \{v(y)\}$ iff $\{v'(x)\} \leq \{v'(y)\}$. The equivalence relation is called the *region equivalence* for the set of clocks X w.r.t. M, and an equivalence class is called a *region*. The set of regions, given X and M, is denoted Reg_M^X. A region r' is a *time-successor* of a region r if there is $v \in r$ and $t \in \mathbb{R}_{\geq 0}$ such that $v + t \in r'$. The set of all time-successors of r is denoted $\overrightarrow{r}$.

In the following, we often abuse notations for guards, invariants, relations and regions, and write g, I, C and r, respectively, for both the constraints over clock variables and the sets of valuations they represent.

9.3.2 Existing approaches to the determinization of TAs

To overcome the non-feasibility of determinization of timed automata in general, two alternatives have been explored: either exhibiting subclasses of timed automata which are determinizable and provide determinization algorithms, or constructing deterministic over-approximations. We relate here, for each of these directions, a recent contribution.

Determinization procedure. An abstract determinization procedure which effectively constructs a deterministic timed automaton for several classes of determinizable timed automata is presented in (3). Given a timed automaton $\mathcal{A}$, this procedure first produces a language-equivalent infinite timed tree, by unfolding $\mathcal{A}$, introducing a fresh clock at each step. This allows one to preserve all timing constraints, using a mapping from clocks of $\mathcal{A}$ to the new clocks. Then, the infinite tree is split into regions, and symbolically determinized. Under a clock-boundedness assumption, the infinite tree with infinitely many clocks can be folded up into a timed automaton (with finitely many locations and clocks). The clock-boundedness assumption is satisfied for several classes of timed automata, such as event-clock TAs (146), TAs with integer resets (147) and strongly non-Zeno TAs (145), which can thus be determinized by this procedure. The resulting deterministic timed automaton is doubly exponential in the size of $\mathcal{A}$.

Deterministic over-approximation By contrast, Krichen and Tripakis propose an algorithm applicable to any timed automaton $\mathcal{A}$, which produces a deterministic over-approximation, that is a deterministic TA B accepting at least all timed words in $\mathcal{L}(\mathcal{A})$ (2). This TA B is built by

simulation of $\mathcal{A}$ using only information carried by clocks of B. A location of B is then a state estimate of $\mathcal{A}$ consisting of a (generally infinite) set of pairs (ℓ, v) where ℓ is a location of $\mathcal{A}$ and v a valuation over the union of clocks of $\mathcal{A}$ and B. This method is based on the use of a fixed finite automaton (the *skeleton*) which governs the resetting policy for the clocks of B. The size of obtained deterministic timed automaton B is also doubly exponential in the size of $\mathcal{A}$.

9.4 A game approach

In (148), given a plant —modeled by a timed automaton— and fixed resources, the authors build a game where some player has a winning strategy if and only if the plant can be diagnosed by a timed automaton using the given resources. Inspired by this construction, given a timed automaton $\mathcal{A}$ and fixed resources, we derive a game between two players Spoiler and Determinizator, such that if Determinizator has a winning strategy, then a deterministic timed automaton B with $\mathcal{L}(\mathsf{B}) = \mathcal{L}(\mathcal{A})$ can be effectively generated. Moreover, any strategy for Determinizator (winning or not) yields a deterministic over-approximation for $\mathcal{A}$. For simplicity, we present here the method for timed automa without ε-transitions and for which all invariants are true.

9.4.1 Definition of the game

Let $\mathcal{A} = (L, \ell_0, F, \Sigma, X, M, E)$ be a timed automaton. We aim at building a deterministic timed automaton B with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathsf{B})$ if possible, or $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathsf{B})$. In order to do so, we fix resources (k, M') for B and build a finite 2-player turn-based safety game $\mathcal{G}_{\mathcal{A},(k,M')}$. Players Spoiler and Determinizator alternate moves, and the objective of player Determinizator is to avoid a set of bad states (to be defined later). Intuitively, in the safe states, for sure, no over-approximation has been performed.

For simplicity, we first detail the approach in the case where $\mathcal{A}$ has no ε-transitions and all invariants are true.

Let Y be a set of clocks of cardinality k. The initial state of the game is a state of Spoiler consisting of location ℓ_0 (initial location of $\mathcal{A}$) together with the simplest relation between X and Y: $\forall\, x \in X, \forall\, y \in Y, x - y = 0$, and a marking $\top$ (no over-approximation was done so far), together with the null region over Y. In each of its states, Spoiler challenges Determinizator by proposing an M'-bounded region r over Y, and an action $a \in \Sigma$. Determinizator answers by deciding the set of clocks $Y' \subseteq Y$ he wishes to reset. The next state of Spoiler contains a region over Y ($r' = r_{[Y' \leftarrow 0]}$), and a finite set of configurations: triples formed of a location of $\mathcal{A}$, a relation between clocks in X and clocks in Y, and a boolean marking ($\top$ or $\bot$). A state

of Spoiler thus constitutes a states estimate of $\mathcal{A}$, and the role of the markings is to indicate whether over-approximations possibly happened. A state of Determinizator is a copy of the preceding states estimate together with the move of Spoiler. Bad states player Determinizator wants to avoid are on the one hand states of the game where all configurations are marked $\bot$ and, on the other hand, states where all final configurations (if any) are marked $\bot$.

Formally, given $\mathcal{A}$ and (k, M') we define $\mathcal{G}_{\mathcal{A},(k,M')} = (\mathsf{V}, \mathsf{v}_0, \mathsf{Ac}, \delta, \mathsf{Bad})$ where:

- The set of vertices V is partitioned into V_S and V_D, respectively vertices of Spoiler and Determinizator. Vertices of V_S and V_D are labeled respectively in $2^{L\times\mathsf{Rel}_{M,M'}(X,Y)\times\{\top,\bot\}} \times \mathsf{Reg}^Y_{M'}$ and $2^{L\times\mathsf{Rel}_{M,M'}(X,Y)\times\{\top,\bot\}} \times (\mathsf{Reg}^Y_{M'} \times \Sigma)$;
- $\mathsf{v}_0 = (\{\overline{0}\}, \{(\ell_0, X - Y = 0, \top)\})$ is the initial vertex and belongs to player Spoiler[1];
- $\mathsf{Ac} = (\mathsf{Reg}^Y_{M'} \times \Sigma) \cup 2^Y$ is the set of possible actions;
- $\delta \subseteq \mathsf{V}_S \times (\mathsf{Reg}^Y_{M'} \times \Sigma) \times \mathsf{V}_D \cup \mathsf{V}_D \times 2^Y \times \mathsf{V}_S$ is the set of edges;
- $\mathsf{Bad} = \{(\{(\ell_j, C_j, \bot)\}_j, r)\} \cup \{(\{(\ell_j, C_j, b_j)\}_j, r) \mid \{\ell_j\}_j \cap F \neq \varnothing \wedge \forall j,\, \ell_j \in F \Rightarrow b_j = \bot\}$ is the set of bad states.

We now detail the edge relation which defines the possible moves of the players. Given $\mathsf{v}_S = (\{(\ell_j, C_j, b_j)\}_j, r) \in \mathsf{V}_S$ a state of Spoiler and (r', a) one of its moves, the successor state is defined, provided r' is a time-successor of r, as the state $\mathsf{v}_D = (\{(\ell_j, C_j, b_j)\}_j, (r', a)) \in \mathsf{V}_D$ if $\exists (\ell, C, b) \in \{(\ell_j, C_j, b_j)\}_j$ and $\exists\, \ell \xrightarrow{g,a,X'} \ell' \in E$ s.t. $[r' \cap C]_{|X} \cap g \neq \varnothing$.

Given $\mathsf{v}_D = (\{(\ell_j, C_j, b_j)\}_j, (r', a)) \in \mathsf{V}_D$ a state of Determinizator and $Y' \subseteq Y$ one of its moves, the successor state of v_D is the state $(\mathcal{E}, r'_{[Y'\leftarrow 0]}) \in \mathsf{V}_S$ where $\mathcal{E}$ is obtained as the set of all elementary successors of configurations in $\{(\ell_j, C_j, b_j)\}_j$ by (r', a) and by resetting Y'. Precisely, if (ℓ, C, b) is a configuration, its elementary successors set by (r', a) and Y' is:

$$\mathsf{Succ}_e[r', a, Y'](\ell, C, b) = \left\{ (\ell', C', b') \;\middle|\; \begin{array}{l} \exists\, \ell \xrightarrow{g,a,X'} \ell' \in E \text{ s.t. } [r' \cap C]_{|X} \cap g \neq \varnothing \\ C' = \mathsf{up}(r', C, g, X', Y') \\ b' = b \wedge ([r' \cap C]_{|X} \cap \neg g = \varnothing) \end{array} \right\}$$

where $\mathsf{up}(r', C, g, X', Y')$ is the update of the relation between clocks in X and Y after the moves of the two players, that is after taking action a in r', resetting $X' \subseteq X$ and $Y' \subseteq Y$, and forcing the satisfaction of g. Formally, $\mathsf{up}(r', C, g, X', Y') = \overleftrightarrow{(r' \cap C \cap g)_{[X'\leftarrow 0][Y'\leftarrow 0]}}$. Boolean b' is set to $\bot$ if either $b = \bot$ or the induced guard $[r' \cap C]_{|X}$ over-approximates g. In the update, the intersection with g aims at stopping runs that for sure will correspond to timed words out

[1] $X - Y = 0$ is a shortcut to denote the relation $\forall\, x \in X, \forall\, y \in Y,\, x - y = 0$.

of $\mathcal{L}(\mathcal{A})$; the boolean b anyway takes care of keeping track of the possible over-approximation. Region r', relation C and guard g can all be seen as zones (*i.e.* unions of regions) over clocks $X \cup Y$. It is standard that elementary operations on zones, such as intersections, resets, future and past, can be performed effectively. As a consequence, the update of a relation can also be computed effectively.

Given the labeling of states in the game $\mathcal{G}_{\mathcal{A},(k,M')}$, the size of the game is doubly exponential in the size of $\mathcal{A}$. We will see in Subsection 9.4.3 that the number of edges in $\mathcal{G}_{\mathcal{A},(k,M')}$ can be impressively decreased, since restricting to atomic resets (resets of at most one clock at a time) does not diminish the power of Determinizator.

As an example, the construction of the game is illustrated on the nondeterministic timed automaton $\mathcal{A}$ depicted in Figure 9.1, for which we construct the associated game $\mathcal{G}_{\mathcal{A},(1,1)}$ represented in Figure 9.2. Rectangular states belong to Spoiler and circles correspond to states of Determinizator. Note that, for the sake of simplicity, the labels of states of Determinizator are omitted in the picture. Gray states form the set Bad. Let us detail the computation of the successors of the top left state by the move $((0,1), b)$ of Spoiler and moves ($\varnothing$ or $\{y\}$) of Determinizator. To begin with, note that b cannot be fired from ℓ_0 in $\mathcal{A}$, therefore the first configuration has no elementary successor. We then consider the configuration which contains the location ℓ_1. The guard induced by $x - y = 0$ and $y \in (0,1)$ is simply $0 < x < 1$ and the guard of the corresponding transition between ℓ_1 and ℓ_3 in $\mathcal{A}$ is exactly $0 < x < 1$, moreover this transition resets x. As a consequence, the successors states contain a configuration marked $\top$ with location ℓ_3 and, respectively, relations $-1 < x - y < 0$ and $x - y = 0$ for the two different moves of Determinizator. Last, when considering the configuration with location ℓ_2, we obtain elementary successors marked $\bot$. Indeed, the guard induced by this move of Spoiler and the relation $-1 < x - y < 0$ is $-1 < x < 1$ whereas the corresponding guard in $\mathcal{A}$ is $x = 0$. To preserve all timed words accepted by $\mathcal{A}$, we represent these configurations, but they imply over-approximations. Thus the successor states contain a configuration marked $\bot$ with location ℓ_3 and the same respective relations as before, thanks to the intersection with the initial guard $x = 0$ in $\mathcal{A}$.

9.4.2 Properties of the strategies

Given $\mathcal{A}$ a timed automaton and resources (k, M'), the game $\mathcal{G}_{\mathcal{A},(k,M')}$ is a finite-state safety game. It is well known that, for this kind of games, winning strategies can be chosen positional and they can be computed in linear time in the size of the arena (149). In the following, we simply write strategies for positional strategies. We will see in Subsection 9.4.3 that positional strategies

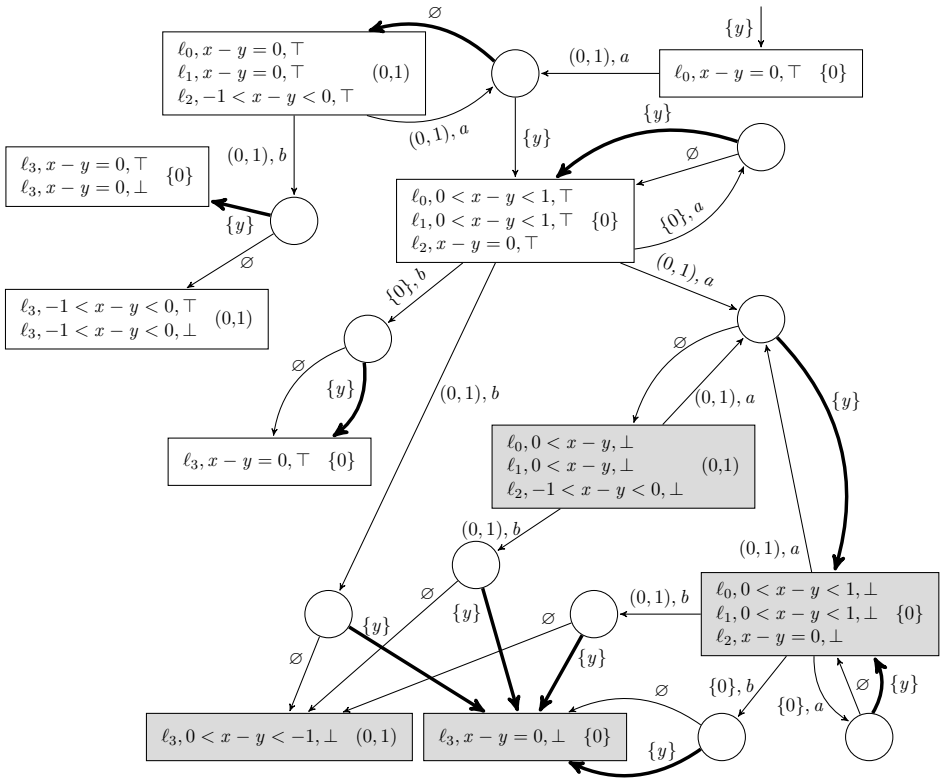


Figure 9.2: The game $\mathcal{G}_{\mathcal{A},(1,1)}$ and an example of winning strategy σ for Determinizator.

(winning or not) are indeed sufficient in our framework. A strategy for player Determinizator thus assigns to each state $\mathsf{v}_D \in \mathsf{V}_D$ a set $Y' \subseteq Y$ of clocks to be reset; the successor state is then $\mathsf{v}_S \in \mathsf{V}_S$ such that $(\mathsf{v}_D, Y', \mathsf{v}_S) \in \delta$.

With every strategy for Determinizator σ we associate the timed automaton $\mathsf{Aut}(\sigma)$ obtained by merging a transition of Spoiler with the transition chosen by Determinizator just after, and setting final locations as states of Spoiler containing at least one final location of $\mathcal{A}$. If a strategy σ_S for Spoiler is fixed too, we denote by $\mathsf{Aut}(\sigma, \sigma_S)$ the resulting sub-automaton[2]. The main result of the chapter is stated in the following theorem and links strategies of Determinizator with deterministic over-approximations of the initial timed language.

Theorem 1 *Let $\mathcal{A}$ a timed automaton, and $k, M' \in \mathbb{N}$. For every strategy σ of Determinizator in $\mathcal{G}_{\mathcal{A},(k,M')}$, $\mathsf{Aut}(\sigma)$ is a deterministic timed automaton over resources (k, M') and satisfies $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathsf{Aut}(\sigma))$. Moreover, if σ is winning, then $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathsf{Aut}(\sigma))$.*

Back to our running example, on Figure 9.2, a winning strategy for Determinizator is represented by the bold arrows. This strategy yields the deterministic equivalent for $\mathcal{A}$ depicted in Figure 9.3.

[2]In the case where σ and/or σ_S have arbitrary memory, we abuse notation and write $\mathsf{Aut}(\sigma)$ and $\mathsf{Aut}(\sigma, \sigma_S)$ for the resulting potentially infinite objects.

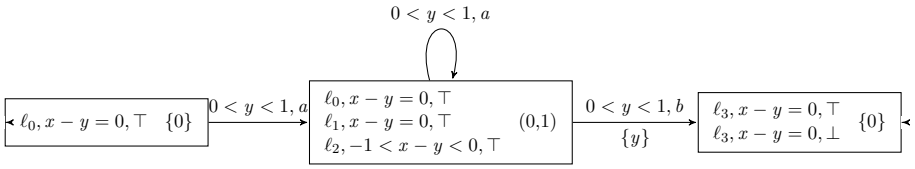


Figure 9.3: The deterministic TA $\mathsf{Aut}(\sigma)$ obtained by our construction.

9.4.3 Choosing a good losing strategy

Standard techniques allow one to check whether there is a winning strategy for Determinizator, and in the positive case, extract such a strategy (149). However, if Determinizator has no winning strategy to avoid the set of bad states, it is of interest to be able to choose a good losing strategy. To this aim, we introduce a natural partial order over the set of strategies of Determinizator based on the distance to the set Bad: $d_{\mathsf{Bad}}(\mathcal{A})$ denotes the minimal number of steps in some automaton $\mathcal{A}$ to reach Bad from the initial state.

Definition 2 *Let σ_1 and σ_2 be strategies of Determinizator in $\mathcal{G}_{\mathcal{A},(k,M')}$. Strategy σ_1 is said* finer *than σ_2, denoted $\sigma_1 \ll \sigma_2$, if for every strategy σ_S of Spoiler, $d_{\mathsf{Bad}}(\mathsf{Aut}(\sigma_1, \sigma_S)) \geq d_{\mathsf{Bad}}(\mathsf{Aut}(\sigma_2, \sigma_S))$.*

Given this definition, an optimal strategy for Determinizator is a minimal element for the partial order $\ll$. Note that, if they exist, winning strategies are the optimal ones since against all strategies of Spoiler, the corresponding distance to Bad is infinite. The set of optimal strategies can be computed effectively by a fix-point computation using a rank function on the vertices of the game.

With respect to this partial order on strategies, positional strategies are sufficient for Determinizator.

Proposition 1 *For every strategy σ of Determinizator with arbitrary memory, there exists a positional strategy σ' such that $\sigma' \ll \sigma$.*

Strategy σ' is obtained from σ by letting for each state the first choice made in σ; this cannot decrease the distance to Bad. Strategies of interest for Determinizator can be even more restricted. Indeed, any timed automaton can be turned into an equivalent one with atomic resets only. Thus, for every strategy for Determinizator there is finer one which resets at most one clock on each transition, which can be turned into a finer positional strategy thanks to Proposition 1. As a consequence, with respect to $\ll$, positional strategies that only allow for atomic resets are sufficient for Determinizator.

9.5 Extension to ε-transitions and invariants

In Section 9.4 the construction of the game and its properties were presented for a restricted class of timed automata. Let us now briefly explain how to extend the previous construction to deal with ε-transitions and invariants. The extension is presented in details in (150).

ε-transitions We aim at building an over-approximation without ε-transitions. An ε-closure is performed for each state during the construction of the game. To this attempt, states of the game have to be extended since ε-transitions might be enabled only from some time-successors of the region associated with the state. Therefore, each configuration is associated with a proper region which is a time-successor of the initial region of the state. The ε-closure is effectively computed the same way as successors in the original construction when Determinizator does not reset any clock; computations thus terminate for the same reasons.

Invariants Ignoring all invariants surely yields an over-approximation. In order to be more precise (while preserving the over-approximation) with each state of the game is associated the most restrictive invariant which contains invariants of all the configurations in the state. In the computation of the successors, invariants are treated similarly to guards and their validity is verified at the transition's target. A state whose invariant is strictly over-approximated is not safe.

9.6 Comparison with existing methods

The method we presented is both more precise than the algorithm of (2) and more general than the procedure of (3). Let us detail these two points. Note that a deeper comparison with existing work can be found in (150).

9.6.1 Comparison with (2)

First of all, our method covers the application area of (2) since each time the latter algorithm produces a deterministic equivalent with resources (k, M') for a timed automaton $\mathcal{A}$, there is a winning strategy for Determinizator in $\mathcal{G}_{\mathcal{A},(k,M')}$.

Moreover, contrary to the method presented in (2), our game-approach is exact on deterministic timed automata: given a DTA $\mathcal{A}$ over resources (k, M), Determinizator has a winning strategy in $\mathcal{G}_{\mathcal{A},(k,M)}$. This is a consequence of the more general fact that, in our approach, a winning strategy can be seen as a timed generalization of the notion of skeleton (2), and solving our game amounts to finding a relevant timed skeleton.

As an example, the algorithm of (2) run on the timed automaton of Figure 9.1 produces a

strict over-approximation, represented on Figure 9.4.

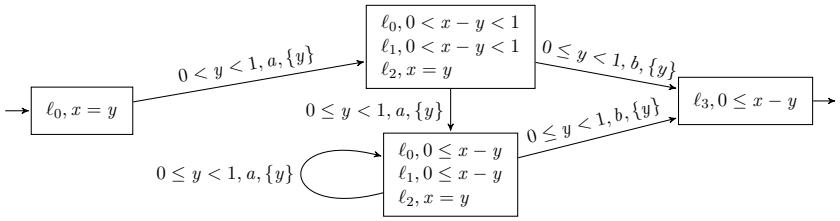


Figure 9.4: The result of algorithm (2) on the running example.

Our approach also improves the updates of the relations between clocks by taking the original guard into account. Precisely, when computing up_S, an intersection with the guard in the original TA is performed. This improvement allows one, even under the same resetting policy, to refine the over-approximation given by (2).

9.6.2 Comparison with (3)

Our approach generalizes the one in (3) since, for any timed automaton $\mathcal{A}$ such that the procedure in (3) yields an equivalent deterministic timed automaton with k clocks and maximal constant M', there is a winning strategy for Determinizator in $\mathcal{G}_{\mathcal{A},(k,M')}$. This can be explained by the fact that relations between clocks of $\mathcal{A}$ and clocks in the game allow one to record more information than the mapping used in (3). Moreover, our approach strictly broadens the class of automata determinized by the procedure of (3) in two respects. First of all, our method allows one to cope with some language inclusions, contrary to (3). For example, the TA depicted on the left-hand side of Figure 9.5 cannot be treated by the procedure of (3) but is easily determinized using our approach. In this example, the language of timed words accepted in location ℓ_3 is not determinizable. This will cause the failure of (3). However, all timed words accepted in ℓ_3 also are accepted in ℓ_4 and the language of timed words accepted in ℓ_4 is clearly determinizable. Our approach allows one to deal with such language inclusions, and will thus provide an equivalent deterministic timed automaton. Second, the relations between clocks of the TA and clocks of the game are more precise than the mapping used in (3), since the mapping can be seen as restricted relations: a conjunction of constraints of the form $x - y = 0$. The precision we add by considering relations rather than mappings is sometimes crucial for the determinization. For example, the TA represented on the right-hand side of Figure 9.5 can be determinized by our game-approach, but not by (3).

Apart from strictly broadening the class of timed automata that can be automatically determinized, our approach performs better on some timed automata by providing a deterministic timed automaton with less resources. This is the case on the running example of Figure 9.1.

Figure 9.5: Examples of determinizable TAs not treatable by (3).

The deterministic automaton obtained by (3) is depicted in Figure 9.6: it needs 2 clocks when our method produces a single-clock TA.

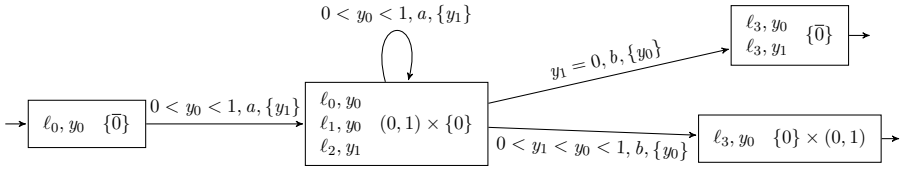


Figure 9.6: The result of procedure (3) on the running example.

The same phenomenon happens with timed automata with integer resets. Timed automata with integer resets, introduced in (147), form a determinizable subclass of timed automata, where every edge (ℓ, g, a, X', ℓ') satisfies $X' \neq \varnothing$ if and only if g contains an atomic constraint of the form $x = c$ for some clock x.

Proposition 2 *For every timed automaton $\mathcal{A}$ with integer resets and maximal constant M, Determinizator has a winning strategy in $\mathcal{G}_{\mathcal{A},(1,M)}$.*

Intuitively, a single clock is needed to represent clocks of $\mathcal{A}$ since they all share a common fractional part.

As a consequence of Proposition 2, any timed automaton with integer resets can be determinized into a doubly exponential single-clock timed automaton with the same maximal constant. This improves the result given in (3) where any timed automaton with integer resets and maximal constant M can be turned into a doubly exponential deterministic timed automaton, using $M + 1$ clocks. Moreover, our procedure is optimal on this class thanks to the lower-bound provided in (151).

Last, our method even when restricted to equality relations (conjunctions of constraints of the form $x - y = c$) extends the procedure of (3). Note that the latter construction is similar to our approach restricted to mappings instead of relations. We detail in (150) the benefits of (even equality) relations and explain how the sufficient conditions for termination provided in (3) can be weakened in our context.

9.6.3 Comparison of the extension with ε-transition and invariants

Let us now compare our extended approach with the approach of (2) since the determinization procedure of (3) does not deal with invariants and ε-transitions.

The model in (2) consists of timed automata with silent transitions and actions are classified depending on their urgency: eager, lazy or delayable. First of all, the authors propose an ε-closure computation which does not terminate in general, and bring up the fact that termination can be ensured by some abstraction. Second, the urgency in the model is not preserved by their over-approximation construction which only produces lazy transitions. Note that we classically decided to use invariants to model urgency, but our approach could be adapted to the same model as the one they use, while preserving urgency much more often, the same way as we do for invariants.

9.7 Summary

In this chapter, we proposed a game-based approach for the determinization of timed automata. Given a timed automaton $\mathcal{A}$ (with ε-transitions and invariants) and resources (k, M), we build a finite turn-based safety game between two players Spoiler and Determinizator, such that any strategy for Determinizator yields a deterministic over-approximation of the language of $\mathcal{A}$ and any winning strategy provides a deterministic equivalent for $\mathcal{A}$. Our construction strictly covers and improves two existing approaches (2; 3).

CHAPTER 10

Off-line Test Selection for Non-Deterministic Timed Automata

10.1 Introduction

This chapter proposes novel off-line test generation techniques for non-deterministic timed automata with inputs and outputs (TAIOs) in the formal framework of the tioco conformance theory. In this context, a first problem is the determinization of TAIOs, which is necessary to foresee next enabled actions, but is in general impossible. This problem is solved here thanks to an approximate determinization using a game approach, which preserves tioco and guarantees the soundness of generated test cases. A second problem is test selection for which a precise description of timed behaviors to be tested is carried out by expressive test purposes modeled by a generalization of TAIOs. Finally, using a symbolic co-reachability analysis guided by the test purpose, test cases are generated in the form of TAIOs equipped with verdicts.

This chapter is structured as follows. In the next section we give some motivation about he proposed approach. In section 10.3m we introduce the model of OTAIOs. Section 10.4 recalls the tioco conformance theory including expected properties relating conformance and verdicts, and an io-refinement relation preserving tioco. Section 10.5 presents our game approach for the approximate determinization compatible with the io-refinement. In Section 10.6 we detail the test selection mechanism using test purposes. Section 10.7 concludes the chapter.

10.2 Motivation

Conformance testing is the process of testing whether an implementation behaves correctly with respect to a specification. Implementations are considered as *black boxes*, *i.e.* the source code is unknown, only their interface with the environment is known and used to interact with the tester. In *formal model-based conformance testing* models are used to describe testing artifacts (specifications, implementations, test cases, ...), conformance is formally defined and test cases with verdicts are generated automatically. Then, the quality of testing may be characterized by properties of test cases which relate the verdicts of their executions with conformance (*e.g.* soundness). For timed models, model-based conformance testing has already been explored in the last decade, with different models and conformance relations (see *e.g.* (152) for a survey), and test generation algorithms (e.g. (153; 2; 154)). In this context, a very popular model is *timed automata with inputs and outputs* (TAIOs), a variant of *timed automata* (TAs) (11), in which observable actions are partitioned into inputs and outputs. We consider here partially observable and non-deterministic TAIOs with invariants for the modeling of urgency.

One of the main difficulties encountered in test generation for TAIOs is determinization, which is impossible in general, as for TAs (11), but is required in order to foresee the next enabled actions during execution and to emit a correct verdict. Two different approaches have been taken for test generation from timed models, which induce different treatments of non-determinism. In *off-line test generation* test cases are first generated as TAs (or timed sequences, trees, or timed transition systems) and subsequently executed on the implementation. Test cases can then be stored and further used e.g. for regression testing and documentation. However, due to the non-determinizability of TAIOs, the approach has often been limited to deterministic or determinizable TAIOs (see e.g. (155; 154)), except in (2) where the problem is solved by the use of an over-approximate determinization with fixed resources, or (156) where winning strategies of timed games are used as test cases. In *on-line test generation*, test cases are generated during their execution, thus can be applied to any TAIO as only possible observable actions are computed along the current finite execution, thus avoiding a complete determinization. This is of particular interest to rapidly discover errors, but may sometimes be impracticable due to a lack of reactivity (the time needed to compute successor states on-line may sometimes be incompatible with delays).

In this work, we propose to generate test cases off-line for non-deterministic TAIOs, in the formal context of the tioco conformance theory. The determinization problem is tackled thanks to an approximate determinization with fixed resources in the spirit of (2), using a game approach (157). Determinization is exact for known classes of determinizable TAIOs (e.g.

event-clock TAs, TAs with integer resets, strongly non-Zeno TAs) if resources are sufficient. In the general case, approximate determinization guarantees soundness of generated test cases by producing a deterministic *io-abstraction* of the TAIO for a particular *io-refinement* relation, generalizing the io-refinement of (158). Our method is more precise than (2) (see (157) for details) and preserves the richness of our model by dealing with partial observability and urgency. Behaviors of specifications to be tested are identified by means of test purposes defined as *open timed automata with inputs and outputs* (OTAIOs), a model generalizing TAIOs, allowing to precisely describe behaviors according to actions and clocks of the specification as well as proper clocks. Then, in the same spirit as for the TGV tool in the untimed case (159), test selection is performed by a co-reachability analysis, producing a test case in the form of a TAIO. To our knowledge, this work constitutes the most general and advanced off-line test selection approach for TAIOs.

10.3 A model of open timed automata with inputs/outputs

We start by introducing notations and definitions concerning TAIOs and OTAIOs.

Given X a finite set of *clocks*, and $\mathbb{R}_{\geq 0}$ the set of non-negative real numbers, a *clock valuation* is a mapping $v : X \to \mathbb{R}_{\geq 0}$. If v is a valuation over X and $t \in \mathbb{R}$, then $v+t$ denotes the valuation which assigns to every clock $x \in X$ the value $v(x) + t$. For $X' \subseteq X$ we write $v_{[X' \leftarrow 0]}$ for the valuation equal to v on $X \setminus X'$ and assigning 0 to all clocks of X'.

Given M a non-negative integer, an *M-bounded guard* (or simply guard) over X is a finite conjunction of constraints of the form $x \sim c$ where $x \in X$, $c \in [0, M] \cap \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$. Given g a guard and v a valuation, we write $v \models g$ if v satisfies g. We abuse notations and write g for the set of valuations satisfying g. *Invariants* are restricted cases of guards: given $M \in \mathbb{N}$, an M-bounded invariant over X is a finite conjunction of constraints of the form $x \lhd c$ where $x \in X, c \in [0, M] \cap \mathbb{N}$ and $\lhd \in \{<, \leq\}$. We denote by $G_M(X)$ (resp. $I_M(X)$) the set of M-bounded guards (resp. invariants) over X.

Definition 3 (OTAIO) *An* open timed automaton with inputs and outputs *(OTAIO) is a tuple* $\mathcal{A} = (L^{\mathcal{A}}, \ell_0^{\mathcal{A}}, \Sigma_?^{\mathcal{A}}, \Sigma_!^{\mathcal{A}}, \Sigma_\tau^{\mathcal{A}}, X_p^{\mathcal{A}}, X_o^{\mathcal{A}}, M^{\mathcal{A}}, \mathsf{Inv}^{\mathcal{A}}, E^{\mathcal{A}})$ *such that:*

- $L^{\mathcal{A}}$ *is a finite set of* locations*, with* $\ell_0^{\mathcal{A}} \in L^{\mathcal{A}}$ *the* initial location*,*
- $\Sigma_?^{\mathcal{A}}$*,* $\Sigma_!^{\mathcal{A}}$ *and* $\Sigma_\tau^{\mathcal{A}}$ *are disjoint finite* alphabets *of* input actions *(noted* $a?, b?, \ldots$*),* output actions *(noted* $a!, b!, \ldots$*), and* internal actions *(noted* $\tau_1, \tau_2, \ldots$*). We note* $\Sigma_{obs}^{\mathcal{A}} = \Sigma_?^{\mathcal{A}} \sqcup \Sigma_!^{\mathcal{A}}$ *(where* $\sqcup$ *denotes the disjoint union) for the alphabet of observable actions, and* $\Sigma^{\mathcal{A}} = \Sigma_?^{\mathcal{A}} \sqcup \Sigma_!^{\mathcal{A}} \sqcup \Sigma_\tau^{\mathcal{A}}$ *for the whole set of actions.*

- $X_p^{\mathcal{A}}$ *and* $X_o^{\mathcal{A}}$ *are disjoint finite sets of* proper clocks *and* observed clocks*, respectively. We note* $X^{\mathcal{A}} = X_p^{\mathcal{A}} \sqcup X_o^{\mathcal{A}}$ *for the whole set of* clocks*.*
- $M^{\mathcal{A}} \in \mathbb{N}$ *is the* maximal constant *of* $\mathcal{A}$*, and we will refer to* $(\mid X^{\mathcal{A}} \mid, M^{\mathcal{A}})$ *as the* resources *of* $\mathcal{A}$*,*
- $\mathsf{Inv}^{\mathcal{A}} : L^{\mathcal{A}} \to I_{M^{\mathcal{A}}}(X^{\mathcal{A}})$ *is a mapping labeling each location with an* invariant*,*
- $E^{\mathcal{A}} \subseteq L^{\mathcal{A}} \times G_{M^{\mathcal{A}}}(X^{\mathcal{A}}) \times \Sigma^{\mathcal{A}} \times 2^{X_p^{\mathcal{A}}} \times L^{\mathcal{A}}$ *is a finite set of* edges *where* guards *are defined on* $X^{\mathcal{A}}$*, but* resets *are restricted to proper clocks in* $X_p^{\mathcal{A}}$*.*

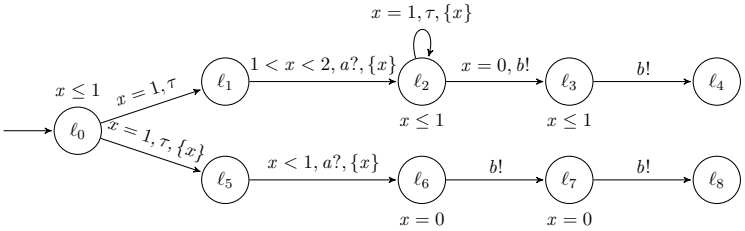


Figure 10.1: Specification $\mathcal{A}$

The reason for introducing the OTAIO model is to have a unique model (syntax and semantics) that will be next specialized for particular testing artifacts. In particular, an OTAIO with an empty set of observed clocks $X_o^{\mathcal{A}}$ is a classical TAIO, and will be the model for specifications, implementations and test cases. For example, Fig. 10.1 represents such a TAIO for a specification $\mathcal{A}$ with clock x, input a, output b and internal action τ. The partition of actions reflects their roles in the testing context: the environment cannot observe internal actions, but controls inputs and observes outputs (and delays). The set of clocks is also partitioned into *proper clocks*, *i.e.* usual clocks controlled by $\mathcal{A}$, and *observed clocks* referring to proper clocks of another OTAIO. These cannot be reset to avoid intrusiveness, but synchronization with them in guards and invariants is allowed. In particular, test purposes have observed clocks which observe proper clocks of specifications in order to describe time constrained behaviors to be tested.

10.4 Conformance testing theory

In this section, we recall the conformance relation tioco (2), that formally defines the set of correct implementations of a given TAIO specification. We then define test cases, formalize their executions, verdicts and expected properties. Finally, we introduce a refinement relation between TAIOs that preserves tioco.

10.4.1 The tioco conformance theory

We consider that the specification is given as a (possibly non-deterministic) TAIO $\mathcal{A} = (L^{\mathcal{A}}, \ell_0^{\mathcal{A}}, \Sigma_?, \Sigma_!, \Sigma_\tau, X_p^{\mathcal{A}}, \varnothing, M^{\mathcal{A}}, \mathsf{Inv}^{\mathcal{A}}, E^{\mathcal{A}})$. The implementation is a black box, unknown except for its alphabet of observable actions, which is the same as the one of $\mathcal{A}$. As usual, in order to formally reason about conformance, we assume that the implementation can be modeled by an (unknown) TAIO $\mathcal{I} = (L^{\Rightarrow}, \ell_0^{\Rightarrow}, \Sigma_?, \Sigma_!, \Sigma_\tau^{\Rightarrow}, X_p^{\Rightarrow}, \varnothing, M^{\Rightarrow}, \mathsf{Inv}^{\Rightarrow}, E^{\Rightarrow})$ with same observable alphabet as $\mathcal{A}$, and require that it is input-complete and non-blocking. The set of such possible implementations of $\mathcal{A}$ is denoted by $\mathcal{I}(\mathcal{A})$. Among these, the conformance relation tioco (2) formally defines which ones conform to $\mathcal{A}$, naturaly extending the ioco relation of Tretmans (160) to timed systems:

Definition 4 (Conformance relation) *Let $\mathcal{A}$ be a TAIO and $\mathcal{I} \in \mathcal{I}(\mathcal{A})$, $\mathcal{I}$ tioco $\mathcal{A}$ if $\forall \sigma \in \mathsf{traces}(\mathcal{A})$, $out(\mathcal{I}\mathsf{after}\sigma) \subseteq out(\mathcal{A}\mathsf{after}\sigma)$.*

Intuitively, $\mathcal{I}$ conforms to $\mathcal{A}$ ($\mathcal{I}\,\mathsf{tioco}\,\mathcal{A}$) if after any timed trace enabled in $\mathcal{A}$, every output or delay of $\mathcal{I}$ is specified in $\mathcal{A}$. In practice, conformance is checked by test cases run on implementations. In our setting, we define test cases as deterministic TAIOs equipped with verdicts defined by a partition of states.

Definition 5 (Test case, test suite) *Given a specification TAIO $\mathcal{A}$, a* test case *for $\mathcal{A}$ is a pair $(\mathcal{TC}, \mathbf{Verdicts})$ consisting of a deterministic TAIO (DTAIO)*
$\mathcal{TC} = (L^{\mathcal{TC}}, \ell_0^{\mathcal{TC}}, \Sigma_?^{\mathcal{TC}}, \Sigma_!^{\mathcal{TC}}, \Sigma_\tau^{\mathcal{TC}}, X_p^{\mathcal{TC}}, \varnothing, M^{\mathcal{TC}}, \mathsf{Inv}^{\mathcal{TC}}, E^{\mathcal{TC}})$ *together with a partition* **Verdicts** *of the set of states $S^{\mathcal{TC}} =$* **None** $\sqcup$ **Inconc** $\sqcup$ **Pass** $\sqcup$ **Fail**. *States outside* **None** *are called* verdict states. *We require that $\Sigma_?^{\mathcal{TC}} = \Sigma_!^{\mathcal{A}}$ and $\Sigma_!^{\mathcal{TC}} = \Sigma_?^{\mathcal{A}}$, $\mathsf{Inv}^{\mathcal{TC}}(\ell) = \mathsf{true}$ for all $\ell \in L^{\mathcal{TC}}$, and $\mathcal{TC}$ is input-complete in all* **None** *states, meaning that it is ready to receive any input from the implementation before reaching a verdict. A* test suite *is a set of test cases.*

The *verdict* of an execution $\sigma \in \mathsf{traces}(\mathcal{TC})$, noted $\mathtt{Verdict}(\sigma, \mathcal{TC})$, is **Pass**, **Fail**, **Inconc** or **None** if $\mathcal{TC}\mathsf{after}\sigma$ is included in the corresponding states set. We note $\mathcal{I}$ `fails` TC if some execution σ of $\mathcal{TC}\|\mathcal{I}$ leads $\mathcal{TC}$ to a **Fail** state, *i.e.* when $\mathsf{traces}_{\mathbf{Fail}}(\mathcal{TC}) \cap \mathsf{traces}(\mathcal{I}) \neq \varnothing$ [1]. Notice that this is only a possibility to reach the **Fail** verdict among the infinite set of executions.

We now introduce soundness, a crucial property ensured by our test generation method and strictness that will be ensured when determinization is exact.

Definition 6 (Test case properties) *A test suite $\mathcal{TS}$ for $\mathcal{A}$ is* sound *if no conformant implementation is rejected by the test suite* i.e. $\forall \mathcal{I} \in \mathcal{I}(\mathcal{A})$, $\forall \mathcal{TC} \in \mathcal{TS}$, $\mathcal{I}$ `fails` $\mathcal{TC} \Rightarrow \neg(\mathcal{I}\,\mathsf{tioco}\,\mathcal{A})$.

[1] The execution of a test case $\mathcal{TC}$ on an implementation $\mathcal{I}$ is usually modeled by the standard parallel composition $\mathcal{TC}\|\mathcal{I}$. Due to space limitations, $\|$ is not defined here, but we use its trace properties: $\mathsf{traces}(\mathcal{I}\|\mathcal{TC}) = \mathsf{traces}(\mathcal{I}) \cap \mathsf{traces}(\mathcal{TC})$.

It is strict *if non-conformance is detected as soon as it occurs* i.e. $\forall \mathcal{I} \in \mathcal{I}(A), \forall \mathcal{TC} \in \mathcal{TS}, \neg(\mathcal{I} \| \mathcal{TC} \text{ tioco } \mathcal{A}) \Rightarrow \mathcal{I} \texttt{ fails } \mathcal{TC}$.

10.4.2 Refinement preserving tioco

We introduce an io-refinement relation between TAIOs, a generalization to non-deterministic TAIOs of the io-refinement between DTAIOs introduced in (158), itself a generalization of alternating simulation (161). We prove that io-abstraction (the inverse relation) preserves tioco: if $\mathcal{I}$ conforms to $\mathcal{A}$, it also conforms to any io-abstraction B of $\mathcal{A}$.

Definition 7 *Let $\mathcal{A}$ and B be two TAIOs with same input and output alphabets, we say that $\mathcal{A}$* io-refines B *(or B* io-abstracts *$\mathcal{A}$) and note $\mathcal{A} \preceq \mathsf{B}$ if*

(i) $\forall \sigma \in \mathsf{traces}(\mathsf{B}),\ out(\mathcal{A}\mathsf{after}\sigma) \subseteq out(\mathsf{B}\mathsf{after}\sigma)$ *and*

(ii) $\forall \sigma \in \mathsf{traces}(\mathcal{A}),\ in(\mathsf{B}\mathsf{after}\sigma) \subseteq in(\mathcal{A}\mathsf{after}\sigma)$.

It can be proved that $\preceq$ is a preorder relation. Moreover, as (ii) is always satisfied if $\mathcal{A}$ is input-complete, for $\mathcal{I} \in \mathcal{I}(\mathcal{A})$, $\mathcal{I}$ tioco $\mathcal{A}$ is equivalent to $\mathcal{I} \preceq \mathcal{A}$. By transitivity of $\preceq$, Proposition 3 states that io-refinement preserves conformance. Its Corollary 1 says that io-abstraction preserves soundness of test suites and will later justify that if a TAIO B io-abstracting $\mathcal{A}$ is obtained by approximate determinization, a sound test suite generated from B is still sound for $\mathcal{A}$.

Proposition 3 *If $\mathcal{A} \preceq \mathsf{B}$ then $\forall \mathcal{I} \in \mathcal{I}(\mathcal{A})$ $(= \mathcal{I}(\mathsf{B}))$, $\mathcal{I}$* tioco *$\mathcal{A} \Rightarrow \mathcal{I}$* tioco *$\mathsf{B}$.*

Corollary 1 *If $A \preceq B$ then any sound test suite for B is also sound for $\mathcal{A}$.*

10.5 Approximate determinization preserving tioco

We recently proposed a game approach to determinize or provide a deterministic over-approximation for TAs (157). Determinization is exact on all known classes of determinizable TAIOs (*e.g.* event-clock TAs, TAs with integer resets, strongly non-Zeno TAs) if resources are sufficient. Provided a couple of extensions, this method can be adapted to the context of testing for building a deterministic io-abstraction of a given TAIO. Thanks to Proposition 3, the construction preserves tioco, and Corollary 1 guarantees the soundness of generated test cases.

The approximate determinization uses the classical region construction (11). As for classical TAs, the regions form a partition of valuations over a given set of clocks which allows to make abstractions and decide properties like the reachability of a location. We note $\mathsf{Reg}_{(X,M)}$ the set of regions over clocks X with maximal constant M. A region r' is a *time-successor* of a region

r if $\exists\, v \in r, \exists\, t \in \mathbb{R}_{\geq 0}, v + t \in r'$. Given X and Y two finite sets of clocks, a *relation* between clocks of X and Y is a finite conjunction C of atomic constraints of the form $x - y \sim c$ where $x \in X,\ y \in Y,\ \sim \in \{<, =, >\}$ and $c \in \mathbb{N}$. When $c \in [-M', M]$, for $M, M' \in \mathbb{N}$, $\mathsf{Rel}_{M,M'}(X, Y)$ we denote the set of relations between X and Y.

10.5.1 A game approach to determinize timed automata

The technique presented in (157) applies first to TAs, *i.e.* the alphabet only consists of one kind of actions (output actions), and the invariants are all trivial. Given such a TA $\mathcal{A}$ over the set of clocks $X^{\mathcal{A}}$, the goal is to build a deterministic TA B with $\mathsf{traces}(\mathcal{A}) = \mathsf{traces}(\mathsf{B})$ as often as possible, or $\mathsf{traces}(\mathcal{A}) \subseteq \mathsf{traces}(\mathsf{B})$. In order to do so, resources of B (number of clocks k and maximal constant $M^{\mathcal{B}}$) are fixed, and a finite 2-player turn-based safety game $\mathcal{G}_{\mathcal{A},(k,M^{\mathcal{B}})}$ is built. The two players, Spoiler and Determinizator, alternate moves, the objective of player Determinizator being to remain in a set of safe states where intuitively, for sure no over-approximation has been performed. Every strategy for Determinizator yields a deterministic automaton B with $\mathsf{traces}(\mathcal{A}) \subseteq \mathsf{traces}(\mathsf{B})$, and every winning strategy induces a deterministic TA B equivalent to $\mathcal{A}$. It is well known that for this kind of games, winning strategies can be chosen positional and computed in linear time in the size of the arena.

Let us now give more details on the definition of the game. Let $X^{\mathcal{B}}$ be a set of clocks of cardinality k. The initial state of the game is a state of Spoiler consisting of the initial location of $\mathcal{A}$, the simplest relation between $X^{\mathcal{A}}$ and $X^{\mathcal{B}}$: $\forall\, x \in X^{\mathcal{A}}, \forall\, y \in X^{\mathcal{B}}, x - y = 0$, a marking $\top$ indicating that no over-approximation was done so far, together with the null region over $X^{\mathcal{B}}$. In each of his states, Spoiler challenges Determinizator by proposing a region $r \in \mathsf{Reg}_{(X^{\mathcal{B}},M^{\mathcal{B}})}$, and an action $a \in \Sigma$. Determinizator answers by deciding the subset of clocks $Y' \subseteq X^{\mathcal{B}}$ he wishes to reset. The next state of Spoiler contains a region over $X^{\mathcal{B}}$ ($r' = r_{[Y' \leftarrow 0]}$), and a finite set of configurations: triples formed of a location of $\mathcal{A}$, a relation between clocks in $X^{\mathcal{A}}$ and clocks in $X^{\mathcal{B}}$, and a boolean marking ($\top$ or $\bot$). A state of Spoiler thus constitutes a states estimate of $\mathcal{A}$, and the role of the markings is to indicate whether over-approximations possibly happened. Bad states Determinizator wants to avoid are states where all configurations are marked $\bot$, *i.e.* configurations where an approximation possibly happened.

A strategy for Determinizator thus assigns to each state of Determinizator a set $Y' \subseteq X^{\mathcal{B}}$ of clocks to be reset. With every strategy for Determinizator Π we associate the TA $\mathsf{B} = \mathsf{Aut}(\Pi)$ obtained by merging a transition of Spoiler with the transition chosen by Determinizator just after. The following theorem links strategies of Determinizator with deterministic over-approximations of the original traces language and enlightens the interest of the game:

Theorem 2 ((157)) *Let $\mathcal{A}$ be a TA, $k, M^{\mathcal{B}} \in \mathbb{N}$. For any strategy Π of Determinizator in $\mathcal{G}_{\mathcal{A},(k,M^{\mathcal{B}})}$, $\mathsf{B} = \mathsf{Aut}(\Pi)$ is a deterministic TA over resources $(k, M^{\mathcal{B}})$ with $\mathsf{traces}(\mathcal{A}) \subseteq \mathsf{traces}(\mathsf{B})$. Moreover, if Π is winning, $\mathsf{traces}(\mathcal{A}) = \mathsf{traces}(\mathsf{B})$.*

10.5.2 Extensions to TAIOs and adaptation to tioco

In the context of model-based testing, the above-mentioned determinization technique must be adapted to TAIOs, as detailed in (157), and summarized below. First the model of TAIOs is more expressive than TAs, incorporating internal actions and invariants. Second, inputs and outputs must be treated differently in order to build from a TAIO $\mathcal{A}$ a DTAIO B such that $\mathcal{A} \preceq \mathsf{B}$ and then preserve tioco.

Internal actions: Specifications naturally include internal actions that cannot be observed during test executions, and should thus be removed during determinization. In order to do so, a closure by internal actions is performed for each state during the construction of the game. To this attempt, states of the game have to be extended since internal actions might be enabled only from some time-successor of the region associated with the state. Therefore, each configuration is associated with a proper region which is a time-successor of the initial region of the state. The closure by silent transitions is effectively computed the same way as successors in the original construction when Determinizator does not reset any clock, computations thus terminate for the same reasons. It is well known that TAs with silent transitions are strictly more expressive than standard TAs (162). Therefore, our approximation can be coarse, but it performs as well as possible with its available clock information.

Invariants: Modeling urgency is quite important and using invariants to this aim is classical. Without the ability to express urgency, for instance, any inactive system would conform to all specifications. Ignoring all invariants in the approximation surely yields an io-abstraction: delays (considered as outputs) are over-approximated. In order to be more precise while preserving $\preceq$, with each state of the game is associated the most restrictive invariant containing invariants of all the configurations in the state. In the computation of the successors, invariants are treated as guards and their validity is verified at both extremities of the transition. A state whose invariant is strictly over-approximated is unsafe.

io-abstraction *vs.* over-approximation: Rather than over-approximating a given TAIO $\mathcal{A}$, we aim here at building a DTAIO B io-abstracting $\mathcal{A}$ ($\mathcal{A} \preceq \mathsf{B}$). Successors by output are over-approximated as in the original game, while successors by inputs must be under-approximated. The over-approximated closure by silent transitions is not suitable to under-approximation. Therefore, states of the game are extended to contain both over- and under-approximated clo-

sures. Thus, the unsafe successors by an input are not built.

All in all, these modifications allow to deal with the full TAIO model with invariants, silent transitions and inputs/outputs, consistently with the io-abstraction. Fig.10.4 represents a part of this game for the TAIO of Fig.10.3. The new game then enjoys the following nice property:

Proposition 4 ((157)[2]) *Let $\mathcal{A}$ be a TAIO, $k, M^{\mathcal{B}} \in \mathbb{N}$. For any strategy Π of Determinizator in $\mathcal{G}_{\mathcal{A},(k,M^{\mathcal{B}})}$, $\mathsf{B} = \mathsf{Aut}(\Pi)$ is a DTAIO over resources $(k, M^{\mathcal{B}})$ with $\mathcal{A} \preceq \mathsf{B}$. Moreover, if Π is winning, $\mathsf{traces}(\mathcal{A}) = \mathsf{traces}(\mathsf{B})$.*

In other words, the approximations produced by our method are deterministic io-abstractions of the initial specification, hence our approach preserves tioco (Proposition 3) and soundness of test cases (Corollary 1). In comparison, the algorithm proposed in (2) is an over-approximation, thus preserves tioco only if the specification is input-complete. Moreover it does not preserve urgency.

10.6 Off-line test case generation

In this section we first define test purposes and then give the principles for off-line test selection with test purposes and properties of generated test cases.

10.6.1 Test purposes

Test purposes are practical means to select behaviors to be tested, either focusing on usual behaviors, or on suspected errors in implementations. In this work we choose the following definition, and discuss alternatives in the conclusion.

Definition 8 (Test purpose) *For a specification TAIO $\mathcal{A}$, a* test purpose *is a pair $(\mathcal{TP}, \mathsf{Accept})$ where $\mathcal{TP} = (L^{\mathcal{TP}}, \ell_0^{\mathcal{TP}}, \Sigma_?, \Sigma_!, \Sigma_\tau, X_p^{\mathcal{TP}}, X_o^{\mathcal{TP}}, M^{\mathcal{TP}}, \mathsf{Inv}^{\mathcal{TP}}, E^{\mathcal{TP}})$ is a complete OTAIO (in particular $\forall \ell \in L^{\mathcal{TP}}, \mathsf{Inv}^{\mathcal{TP}}(\ell) = \mathsf{true}$) with $X_o^{\mathcal{TP}} = X_p^{\mathcal{A}}$ (TP observes proper clocks of $\mathcal{A}$), and $\mathsf{Accept} \subseteq L^{\mathcal{TP}}$ is a subset of trap locations.*

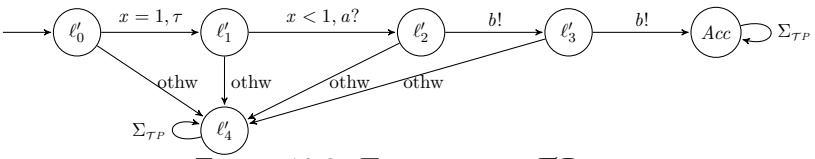


Figure 10.2: Test purpose $\mathcal{TP}$.

Fig. 10.2 represents a test purpose for the specification $\mathcal{A}$ of Fig. 10.1. It has no proper clock and observes the unique clock x of $\mathcal{A}$. It accepts sequences where τ occurs at $x = 1$, followed by an input $a?$ at $x < 1$ (thus focusing on the lower branch of $\mathcal{A}$ where x is reset), and

two subsequent $b!$'s. The label *othw* (for otherwise) is an abbreviation for the complement of specified transitions.

10.6.2 Principle of test generation

Given a specification TAIO $\mathcal{A}$ and a test purpose $(\mathcal{TP}, \mathsf{Accept}^{\mathcal{TP}})$, the aim is to build a sound and, if possible strict test case $(\mathcal{TC}, \mathbf{Verdicts})$. It should also deliver **Pass** verdicts on traces of sequences of $\mathcal{A}$ accepted by $\mathcal{TP}$, as formalized by the following property:

Definition 9 *A test suite $\mathcal{TS}$ for $\mathcal{A}$ and $\mathcal{TP}$ is* precise *if* $\forall \mathcal{TC} \in \mathcal{TS}, \forall \sigma \in (\Sigma^{\mathcal{A}}_{obs})^*, \mathtt{Verdict}(\sigma, \mathcal{TC}) = \mathbf{Pass} \Leftrightarrow \sigma \in \mathsf{traces}(seq^{\mathcal{TP}}_{\mathsf{Accept}}(\mathcal{TP}) \cap seq(\mathcal{A}))$.

The different steps of test generation are described in the following paragraphs.

Product: we first build the TAIO $\mathcal{P} = \mathcal{A} \times \mathcal{TP}$ associated with the set of marked locations $\mathsf{Accept}^{\mathcal{P}} = L^{\mathcal{A}} \times \mathsf{Accept}^{\mathcal{TP}}$. Fig. 10.3 represents this product $\mathcal{P}$ for the specification $\mathcal{A}$ in Fig. 10.1 and the test purpose $\mathcal{TP}$ in Fig. 10.2. The effect of the product is to unfold $\mathcal{A}$ and to mark those sequences of $\mathcal{A}$ accepted by $\mathcal{TP}$ in locations $\mathsf{Accept}^{\mathcal{TP}}$. $\mathcal{TP}$ is complete, thus $\mathrm{seq}(\mathcal{P}) = \mathrm{seq}(\mathcal{A}\uparrow^{X_p^{\mathcal{TP}}, X_o^{\mathcal{TP}}})$ (sequences of the product are sequences of $\mathcal{A}$ lifted to $X^{\mathcal{TP}}$), and then $\mathsf{traces}(\mathcal{P}) = \mathsf{traces}(\mathcal{A})$, which implies that $\mathcal{P}$ and $\mathcal{A}$ define the same sets of conformant implementations. We also have $\mathrm{seq}_{\mathsf{Accept}^{\mathcal{P}}}(\mathcal{P}) = \mathrm{seq}(\mathcal{A}\uparrow^{X_p^{\mathcal{TP}};\, X_o^{\mathcal{TP}}}) \cap \mathrm{seq}_{\mathsf{Accept}^{\mathcal{TP}}}(\mathcal{TP})$ which induces $\mathsf{traces}_{\mathsf{Accept}^{\mathcal{P}}}(\mathcal{P}) = \mathsf{traces}(\mathrm{seq}(\mathcal{A}) \cap \mathrm{seq}_{\mathsf{Accept}^{\mathcal{TP}}}(\mathcal{TP}))$.

Let $\mathsf{ATraces}(\mathcal{A}, \mathcal{TP}) = \mathsf{traces}_{\mathsf{Accept}^{\mathcal{P}}}(\mathcal{P})$ and $\mathsf{RTraces}(\mathcal{A}, \mathcal{TP}) = \mathsf{traces}(\mathcal{A}) \setminus pref(\mathsf{ATraces}(\mathcal{A}, \mathcal{TP}))$ where, for a set of traces T, $pref(T)$ denotes the set of prefixes of traces in T. The principle is to select traces in $\mathsf{ATraces}(\mathcal{A}, \mathcal{TP})$ and try to avoid or at least detect those in $\mathsf{RTraces}(\mathcal{A}, \mathcal{TP})$ as these traces cannot be prefixes of traces of sequences satisfying the test purpose.

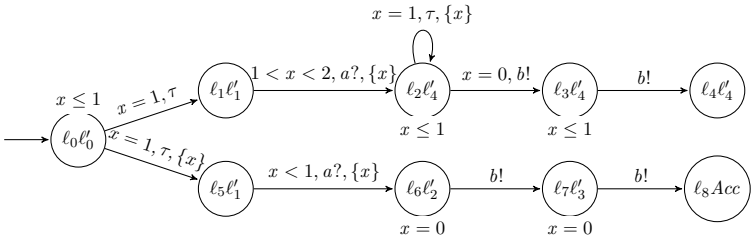


Figure 10.3: Product $\mathcal{P} = \mathcal{A} \times \mathcal{TP}$.

Approximate determinization of $\mathcal{P}$ into $\mathcal{DP}$: If $\mathcal{P}$ is already deterministic, we simply take $\mathcal{DP} = \mathcal{P}$. Otherwise, with the approximate determinization of Section 10.5, we can build a deterministic io-abstraction $\mathcal{DP}$ of $\mathcal{P}$ with resources $(k, M^{\mathcal{DP}})$ fixed by the user, thus $\mathcal{P} \preceq \mathcal{DP}$. $\mathcal{DP}$ is equipped with the set of marked locations $\mathsf{Accept}^{\mathcal{DP}}$ consisting of locations in $L^{\mathcal{DP}}$ containing some configuration whose location is in $\mathsf{Accept}^{\mathcal{P}}$. If the determinization is exact,

we get $\mathsf{traces}(\mathcal{DP}) = \mathsf{traces}(\mathcal{P})$ and $\mathsf{traces}_{\mathsf{Accept}^{\mathcal{DP}}}(\mathcal{DP}) = \mathsf{ATraces}(\mathcal{A}, \mathcal{TP})$. Fig. 10.4 partially represents the game $\mathcal{G}_{\mathcal{P},(1,2)}$ for the TAIO $\mathcal{P}$ of Fig. 10.3 where, for readability reasons, some behaviors not co-reachable from $\mathsf{Accept}^{\mathcal{DP}}$ are omitted. $\mathcal{DP}$ is simply obtained from $\mathcal{G}_{\mathcal{P},(1,2)}$ by merging transitions of Spoiler and Determinizator.

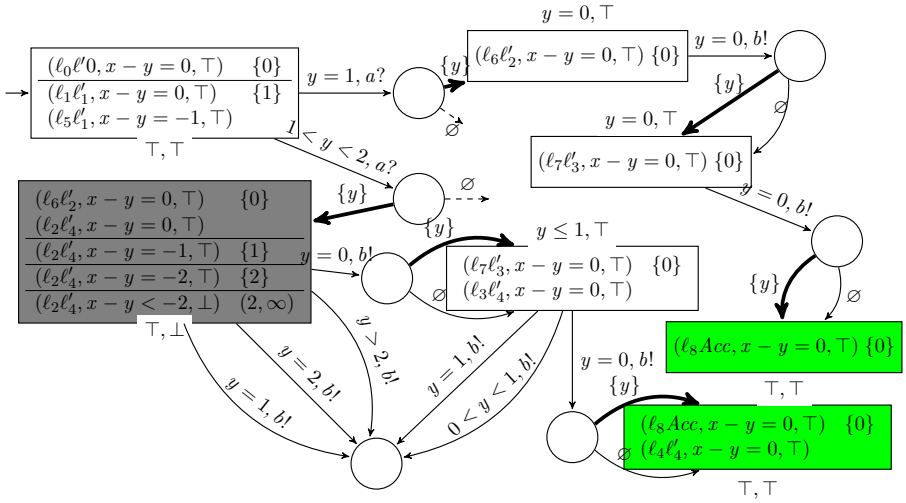


Figure 10.4: Game $\mathcal{G}_{\mathcal{P},(1,2)}$.

Generating $\mathcal{TC}$ from $\mathcal{DP}$: The next step consists in building $(\mathcal{TC}, \textbf{Verdicts})$ from $\mathcal{DP}$, using an analysis of the co-reachability to locations $\mathsf{Accept}^{\mathcal{DP}}$ in $\mathcal{DP}$.

The test case built from $\mathcal{DP} = (L^{\mathcal{DP}}, \ell_0^{\mathcal{DP}}, \Sigma_?^{\mathcal{DP}}, \Sigma_!^{\mathcal{DP}}, X_p^{\mathcal{DP}}, \varnothing, M^{\mathcal{DP}}, \mathsf{Inv}^{\mathcal{DP}}, E^{\mathcal{DP}})$ and $\mathsf{Accept}^{\mathcal{DP}}$ is the TAIO $\mathcal{TC} = (L^{\mathcal{TC}}, \ell_0^{\mathcal{TC}}, \Sigma_?^{\mathcal{TC}}, \Sigma_!^{\mathcal{TC}}, X_p^{\mathcal{TC}}, \varnothing, M^{\mathcal{TC}}, \mathsf{Inv}^{\mathcal{TC}}, E^{\mathcal{TC}})$ such that $L^{\mathcal{TC}} = L^{\mathcal{DP}} \sqcup \{\ell_{\textbf{Fail}}\}$ where $\ell_{\textbf{Fail}}$ is a new location; $\ell_0^{\mathcal{TC}} = \ell_0^{\mathcal{DP}}$; $\Sigma_?^{\mathcal{TC}} = \Sigma_!^{\mathcal{DP}} = \Sigma_!^{\mathcal{A}}$ and $\Sigma_!^{\mathcal{TC}} = \Sigma_?^{\mathcal{DP}} = \Sigma_?^{\mathcal{A}}$, *i.e.* input/output alphabets are mirrored in order to reflect the opposite role of actions in the synchronization of $\mathcal{TC}$ and $\mathcal{I}$; $X_p^{\mathcal{TC}} = X_p^{\mathcal{DP}}$ and $X_o^{\mathcal{TC}} = \varnothing$; $M^{\mathcal{TC}} = M^{\mathcal{DP}}$; **Verdicts** is the partition of $S^{\mathcal{TC}}$ with $\textbf{Pass} = \bigcup_{\ell \in \mathsf{Accept}^{\mathcal{DP}}} \{\ell\} \times \mathsf{Inv}^{\mathcal{DP}}(\ell)$, $\textbf{None} = \mathtt{coreach}(\mathcal{DP}, \textbf{Pass}) \setminus \textbf{Pass}$, $\textbf{Inconc} = S^{\mathcal{DP}} \setminus \mathtt{coreach}(\mathcal{DP}, \textbf{Pass})$, and $\textbf{Fail} = \{\ell_{\textbf{Fail}}\} \times \mathbb{R}_+^{X^{\mathcal{TC}}} \sqcup \{(\ell, \neg \mathsf{Inv}^{\mathcal{DP}}(\ell)) \mid \ell \in L^{\mathcal{DP}}\}$; $\mathsf{Inv}^{\mathcal{TC}}(\ell) = \mathsf{true}$ for any $\ell \in L^{\mathcal{TC}}$; $E^{\mathcal{TC}} = E^{\mathcal{DP}}_{\mathsf{Inv}} \sqcup E_{\ell_{\textbf{Fail}}}$ where $E^{\mathcal{DP}}_{\mathsf{Inv}} = \{(\ell, g \wedge \mathsf{Inv}^{\mathcal{DP}}(\ell), a, X, \ell') \mid (\ell, g, a, X, \ell') \in E^{\mathcal{DP}}\}$ and $E_{\ell_{\textbf{Fail}}} = \{(\ell, \bar{g}, a, X_p^{\mathcal{TC}}, \ell_{\textbf{Fail}}) \mid \ell \in L^{\mathcal{DP}}, a \in \Sigma_!^{\mathcal{DP}}, \bar{g} = \neg \bigvee_{(\ell,g,a,X,\ell') \in E^{\mathcal{DP}}} g\}$.

The important points to understand in the construction of $\mathcal{TC}$ are the completion to **Fail** and the computation of **Inconc**. For the completion, the idea is to detect unspecified outputs and delays of $\mathcal{DP}$. Outputs of $\mathcal{DP}$ being inputs of $\mathcal{TC}$, in any location ℓ, for each input $a \in \Sigma_?^{\mathcal{TC}} = \Sigma_!^{\mathcal{DP}}$, a transition leading to $\ell_{\textbf{Fail}}$ is added, labeled with a, and whose guard is the negation of the disjunction of all guards of transitions labeled by a and leaving ℓ (thus true if no a-action leaves ℓ). Authorized delays in $\mathcal{DP}$ being defined by invariants, all states in $(\ell, \neg \mathsf{Inv}^{\mathcal{DP}}(\ell)), \ell \in L^{\mathcal{DP}}$, *i.e.* states where the invariant runs out, are put into **Fail**. Moreover, in each location ℓ, the

invariant $\mathsf{Inv}^{\mathcal{DP}}(\ell)$ in $\mathcal{DP}$ is removed and shifted to guards of all transitions leaving ℓ in $\mathcal{TC}$.

The computation of **Inconc** is based on an analysis of the co-reachability to **Pass**. **Inconc** contains all states not co-reachable from locations in **Pass**. Notice that $\texttt{coreach}(\mathcal{DP}, \mathbf{Pass})$, and thus **Inconc**, can be computed symbolically in the region graph of $\mathcal{DP}$. Fig.10.5 represents the test case obtained from $\mathcal{A}$ and $\mathcal{TP}$.

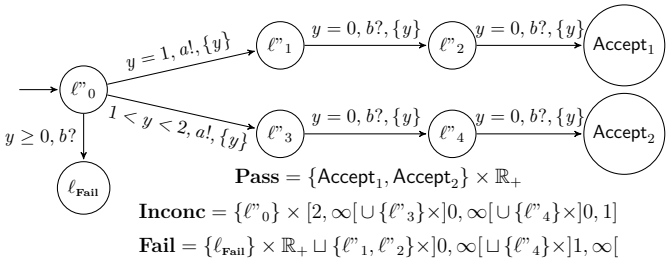


Figure 10.5: Test case $\mathcal{TC}$

Test selection: So far, the construction of $\mathcal{TC}$ determines **Verdicts**, but does not perform any selection of behaviors. A last step consists in trying to control the behavior of $\mathcal{TC}$ in order to avoid **Inconc** states (thus stay in $pref(\mathsf{ATraces}(\mathcal{A}, \mathcal{TP}))$), or produce an **Inconc** verdict when this is impossible. To this aim, guards of transitions are refined in two complementary ways. First, transitions leaving a verdict state are useless, thus for each transition, the guard is intersected with the set of valuations associated with **None** in the source location. Second, transitions arriving in **Inconc** states and carrying inputs are also useless, thus for any transition labeled by an input, the guard is intersected with the set of valuations associated with $\texttt{coreach}(\mathcal{DP}, \mathbf{Pass})$ in the target location. For example in $\mathcal{TC}$ (Fig. 10.5), the bottom-left state of the game in Fig. 10.4 has been removed.

After these steps, generated test cases exhibit the following properties:

Theorem 3 *Any test case $\mathcal{TC}$ built by the procedure is sound for $\mathcal{A}$. If $\mathcal{DP}$ is an exact approximation of $\mathcal{P}$, $\mathcal{TC}$ is also strict and precise for $\mathcal{A}$ and $\mathcal{TP}$.*

Soundness comes from the construction of $E_{\mathbf{Fail}}$ in $\mathcal{TC}$ and preservation of soundness by the approximate determinization $\mathcal{DP}$ of $\mathcal{P}$ given by Corollary 1. When $\mathcal{DP}$ is an exact determinization of $\mathcal{P}$, $\mathsf{traces}(\mathcal{DP}) = \mathsf{traces}(\mathcal{P}) = \mathsf{traces}(\mathcal{A})$. Strictness then comes from the fact that $\mathcal{DP}$ and $\mathcal{A}$ have the same non-conformant traces and from the definition of $E_{\mathbf{Fail}}$ in $\mathcal{TC}$. Precision comes from $\mathsf{traces}_{\mathsf{Accept}^{\mathcal{DP}}}(\mathcal{DP}) = \mathsf{ATraces}(\mathcal{A}, \mathcal{TP})$ and from the definition of **Pass**. When $\mathcal{DP}$ is not exact however, there is a risk that some behaviors allowed in $\mathcal{DP}$ are not in $\mathcal{P}$, thus some non-conformant behaviors are not detected, even if they are executed by $\mathcal{TC}$. Similarly, some **Pass** verdicts may be produced for non-accepted or non-conformant behaviors.

Test execution After test selection, it remains to execute test cases on a real implementation. As the test case is a TAIO, a number of decisions still need to be made at each node of the test case: (1) whether to wait for a certain delay, to receive an input or emit an output (2) which output to send, in case there is a choice. Some of these choices can be made either randomly, or according to user-defined strategies, for example by applying a technique similar to the control approach of (156) whose goal is to avoid $\mathsf{RTraces}(\mathcal{A}, \mathcal{TP})$.

10.7 Summary

In this chapter, we presented a complete formalization and operations for the automatic off-line generation of test cases from non-deterministic timed automata with inputs and outputs (TAIOs). The model of TAIOs is general enough to take into account non-determinism, partial observation and urgency. One main contribution is the ability to tackle any TAIO, thanks to an original approximate determinization procedure. Another main contribution is the selection of test cases with expressive OTAIOs test purposes, able to precisely select behaviors based on clocks and actions of the specification as well as proper clocks. Test cases are generated as TAIOs using a symbolic co-reachability analysis of the observable behaviors of the specification guided by the test purpose.

Part IV

Ongoing Works

CHAPTER 11

Towards a Model-Based Testing Framework for the Security of Internet of Things for Smart City Applications

11.1 Introduction

This chapter reports on a work in progress in which we are interested in testing security aspects of Internet of Things for Smart Cities. For this purpose we follow a Model-Based approach which consists in: modeling the system under investigation with an appropriate formalism; deriving test suites from the obtained model; applying some coverage criteria to select suitable tests; executing the obtained tests; and finally collecting verdicts and analyzing them in order to detect errors and repair them.

The rest of this chapter is organized as follows. Section 11.2 gives the main motivation of the work presented in this chapter. Section 11.3 introduces some preliminaries about IoT and smart cities. Section 11.4 discusses main threats and challenges related to these two fields. Section 11.5 presents our approach. Section 11.6 reports on related research efforts dealing with IoT security testing. Finally Section 11.7 concludes the chapter.

11.2 Motivation

Internet of Things (IoT) is a promising technology that permits to connect everyday things" or objects to the Internet by giving them the capabilities to sense the environment and interact with other objects and/or human beings through the Internet. This evolving technology has promoted a new generation of innovative and valuable services. Today's cities are getting smarter

by deploying intelligent systems for traffic control, water management, energy management, public transport, street lighting, etc. thanks to these services. Nevertheless, these services can easily be compromised and attacked by malicious parties in the absence of proper mechanism for providing adequate security. Recent studies have shown that the attackers are using smart home appliances to launch serious attacks such as infiltrating to the network or sending malicious email or launching malicious actions such as Distributed Denial of Service (DDoS) attack. Therefore, security solutions need to be proposed, set up and tested to mitigate these identified attacks.

In this work, we aim to adopt a Model-Based Security Testing (MBST) approach to check the security of IoT applications in the context of smart cities. The MBST approach consists in specifying the desired IoT application in an abstract manner using an adequate formal specification language and then deriving test-suites from this specification to find security vulnerabilities in the application under test in a systematic manner. The work introduced here is a piece of a broader approach dealing with the security of IoT applications for smart cities and consisting of the following steps:

- Identify and assess the threats and the attacks in smart cities IoT applications.
- Design and develop security mechanisms for standard protocols at the application and the network layer.
- Evaluate the performance and the correctness of the proposed security protocols using simulation and implementation on real devices.

11.3 Preliminaries

11.3.1 Internet of Objects

Recent advances in communication and sensing devices make our everyday objects smarter. This smartness is resulted from the capability of objects to sense the environment, to process the captured (sensed) data and to communicate it to users either directly or through Internet. The integration of these smart objects to the Internet infrastructure is promoting a new generation of innovative and valuable services for people. These services include home automation, traffic control, public transportation, smart water metering, waste and energy management, etc. When integrated in a city context, they make citizens' live better and so form the modern smart city.

11.3.2 Smart Cities

In October 2015, ITU-T's Focus Group on Smart Sustainable Cities (FG-SSC) agreed on the following definition of a smart sustainable city: "A Smart Sustainable City (SSC) is an innovative city that uses information and communication technologies (ICTs) and other means to improve quality of life, efficiency of urban operation and services, and competitiveness, while ensuring that it meets the needs of present and future generations with respect to economic, social and environmental aspects".

11.4 Threats and challenges

11.4.1 Threats

Indeed, connecting our everyday âœthingsâ to the public Internet opens these objects to several kinds of attacks. Taking the example of a traffic control system. If the hackers could insert fake messages to these traffic control system devices, they can make traffic perturbations and bottlenecks. Another example related to home automation, if attackers gain access to smart devices such as lamps, doors, etc., it could manipulate doors and steal the house properties. The main security threats in the IoT are summarized in [28] and they can be summarized as follows: 1. Cloning of smart things by untrusted manufacturers; 2. Malicious substitution of smart things during installation; 3. Firmware replacement attack; 4. Extraction of security parameters since smart things may be physically unprotected; 5. Eavesdropping attack if the communication channel is not adequately protected; 6. Man-in-the-middle attack during key exchange; 7. Routing attacks; 8. Denial-of-service attacks; and 9. Privacy threats.

11.4.2 Challenges

Due to its specific characteristic, new issues are raised in the area of IoT:

- Data collection trust: If the huge collected data is not trusted (e.g., due to the damage or malicious input of some sensors), the IoT service quality will be greatly influenced and hard to be accepted by users.

- User privacy: In order to have intelligent context-aware services, users have to share their personal data or privacy such as location, contacts, etc. Providing intelligent context-aware services and at the same time preserving user privacy are two conflicting objectives that induce a big challenge in the IoT.

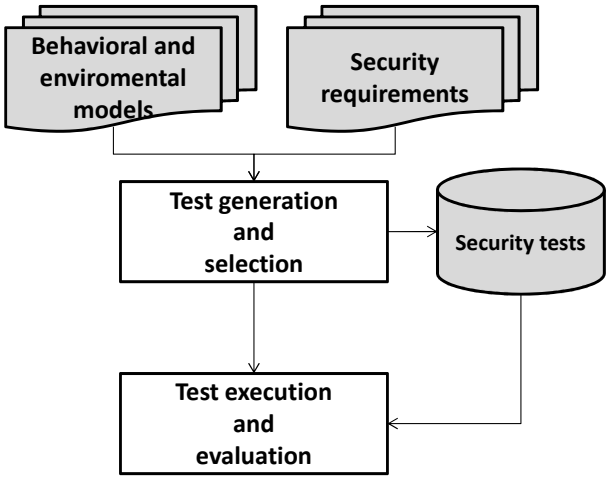


Figure 11.1: Model based security testing process.

- Resource Limitation: Most of IoT devices are limited in terms of CPU, memory capacity and battery supply. This renders the application of the conventional Internet security solutions not appropriate.

- Inherent complexity of IoT: the fact that multiple heterogeneous entities located in different contexts can exchange information with each other, further complicates the design and the deployment of efficient, inter-operable and scalable security mechanisms.

11.5 Proposed Approach

In this section, we define a workflow that covers the different steps of a classical model based testing process, namely: Model Specification, test generation, test selection, test execution and evaluation activities as depicted in Fig. 11.1.

In this direction, we reuse the finding of Hessel et al.(163) by exploiting its extension of UPPAAL namely UPPAAL CO$\sqrt{}$ER . This tool takes as inputs a model, an observer and a configuration file. The model is specified as a network of timed automata (.xml) that comprises a SUT part and an environment part. The observer (.obs) expresses the coverage criterion that guides the model exploration during test case generation. The configuration file (.cfg) describes mainly the interactions between the system part and the environment part in terms of input/output signals. It may also specify the variables that should be passed as parameters in these signals. As output, it produces a test suite containing a set of timed traces (.xml).

Our test generation module is built upon these well-elaborated tools. The key idea here is to use UPPAAL CO$\sqrt{}$ER and its generic and formal specification language for coverage criteria to generate tests for security purposes.

11.5.1 Test Execution and Verdict Analysis

For the execution of the obtained security tests, we aim to use a standard-based test execution platform, called TTCN-3 test system for Runtime Testing (TT4RT), developed in a previous work (164). To do so, security tests should be mapped to the TTCN-3 notation since our platform supports only this test language. Then, test components are dynamically created and assigned to execution nodes in a distributed manner.

Each test component is responsible for (1) stimulating the SUT with input values, (2) comparing the obtained output data with the expected results (also called oracle) and (3) generating the final verdict. The latter can be pass, fail or inconclusive. A pass verdict is obtained when the observed results are valid with respect to the expected ones. A fail verdict is obtained when at least one of the observed results is invalid with respect to the expected one. Finally, an inconclusive verdict is obtained when neither a pass or a fail verdict can be given. After computing for each executed test case its single verdict, the proposed platform deduces the global verdict.

11.6 Related Work

In this section we give a very brief overview on contributions form the literature and from our previous work related to Model-Based Security Testing (MBST) for IoT Applications in Smart Cities. Authors of (165) propose a good survey on more than one hundred publications on model-based security testing extracted from the most relevant digital libraries and classified according to specific criteria. Even though this survey reports on a large number of articles about MBST it does not contain any reference to IoT applications or Smart Cities. Contrary to that the authors of (166) propose a model-based approach to test IoT platforms (with tests provided as services) but they do not deal with security aspects at all.

In this work we aim to combine these two directions namely: Model-Based testing and Security Testing for IoT applications in Smart Cities. For that purpose we will take advantage of our previous findings (167; 168; 164; 169) related to these fields. In (167) a survey about Secure Group Communication in Wireless Sensor Networks is proposed. We will extend the notions proposed in this survey to the case of IoT applications. We will also exploit our previous results about test techniques of dynamic distributed systems (168; 164). Finally we will adopt the same methodology as in (169) to combine security and load tests for IoT applications.

11.7 Summary

The work presented in this chapter is at its begining and a lot of efforts are needed at all levels on both theoretical and experimental aspects. First we need to deal with modelling issues. In this respect we need to extend our modelling formalism and to identify the particular elements of IoT applications to model (using extended timed automata). Models must not be big in order to avoid test number explosion. For that purpose we need to keep an acceptable level of abstraction. As a second step we have to adapt our test generation and selection algorithms to take into account security requirements of the applications under test. The new algorithms must be validated theoretically and proved to be correct. In the same manner we need to upgrade our tools to implement the new obtained algorithms. Finally we need to validate our approach with concrete examples with realistic size.

CHAPTER 12

Towards a Scalable Test Execution Platform On the Cloud

12.1 Introduction

Testing large scale systems running in dynamic and distributed environments is a challenging issue. Such a validation activity needs to be handled in a cost effective manner. To do so, this chapter introduces a scalable test environment deployed on the cloud and offers various testing capabilities like automatic test component deployment, test execution and test evaluation. The latter are provided as services following the SOA architecture. A proof-of-concept prototype is developed and deployed on the Google Cloud Platform. It is used to validate an e-Health case study, implemented by using Web service technology.

The rest of this chapter is organized as follows. Section 12.2 gives the main motivation behind the work presented in this chapter. Section 12.3 provides background material and related work on cloud computing and cloud testing. Section 12.4 outlines the proposed approach. Section 12.5 reports on the application of our approach to the e-Health case study. Finally, in Section 12.6, we conclude with a summary of the main contributions, and we identify potential areas of future research.

12.2 Motivation

Due to increasingly software scale and complexity in recent years, test engineers and quality assurance managers faced many difficulties in terms of test time, cost and scale. In fact, managing test generation and selection issues is still a time consuming aspect in the testing process.

Moreover, setting up distributed test environments for test execution and evaluation concerns increases significantly software production costs because we need computational resources not only for the execution of the system under test but also for the support of testing.

In order to encounter such problems, cloud computing is emerging as a new solution to build scalable and dynamic test environment characterized mainly with on-demand resource allocation capabilities. Known in the literature as Testing as-a-Service (TaaS), this innovative concept is considered as a new business model which provides software testing activities in a cloud infrastructure for customers as a service based on their demands. In (170), many benefits of TaaS are identified and listed as below:

- Reduce costs of setting a distributed test environment by effectively using virtualized resources hosted on the cloud platform.
- Adjust dynamically required resources for testing purposes as needed. The pay-as-you-test model is often linked to the elastic aspect of the cloud.
- Provide on-demand testing services such as test case generation, online/offline test execution and test result evaluation, etc.

For this reason, a recent branch of work has attempted to migrate conventional testing services to the cloud (171; 170; 172; 173; 174; 175). They have focused on offering cloud-based test environments with various options such as resource monitoring, static/dynamic virtual machine management, scheduling and dispatching test tasks to the appropriate virtual machines. Up to our knowledge, only (175) has proposed a test support as a service applied for runtime testing of adaptive systems. The latter used replication strategy to apply safe runtime tests with reducing interference risks between test processes and business processes.

12.3 Background and Related Work

This section presents the background that motivates our work and gives an overview about research done in testing cloud-based applications.

12.3.1 Cloud testing

Cloud computing is an emergent paradigm in the distributed computing community. It has been changed the way of obtaining diverse services (such as software and hardware resources, networks, storage, etc.)(176). It is formally defined by U.S. NIST (National Institute of Standards and Technology)(177) as follows :

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Such paradigm has been used in the context of software testing to encounter the lack of resources and the expensiveness of building a distributed test environment during the testing process. As a result, the concept of *Cloud testing* is newly emerging in order to provide cost-effective testing services. According to (170), it refers to testing activities, essentially test generation, test execution and test evaluation on a cloud-based environment. The latter supports on-demand resource allocation to large scale testers whenever and wherever they need by following the pay-per-use business model. Such virtualized and shared resources may reduce effectively the cost of building a distributed test environment.

12.3.2 Testing as-a-Service

Testing as-a-Service (TaaS) is an innovative concept that provides end users with testing services such as test case generation, test execution and test result evaluation. It has been proposed to improve the efficiency of software quality assurance. Notably, it is used for software systems that are remotely deployed in a virtualized runtime environment using shared hardware/software resources, and hosted in a third-party infrastructure (i.e. a cloud) . One of the primary objectives is to reduce the cost of software testing tasks by providing on-demand testing services and also on-demand test environment services.

12.3.3 Related work

Most existing research in Testing as-a-Service paradigm pays more attention to test clouds and cloud-based applications. General topics about cloud testing issues and challenges have been discussed in several research papers (178; 170). Moreover, some works have dealt with proposing their own cloud-based testing architecture to provide cost-effective testing services.

For instance, the work in (171) introduces the design and implementation of a virtual test system, called Vee@Cloud. The proposed prototype offers on-demand test resource allocation with the aim of reducing cost and enhancing the scalability of the test environment. Moreover, it supports the generation of various workload in order to apply efficient load testing. Likewise, the approach in (173; 174) develops a prototype of TaaS on the cloud that helps test engineers in setting up a scalable test environment for automatically generating and executing units tests. The obtained results are then evaluated and reported to testers.

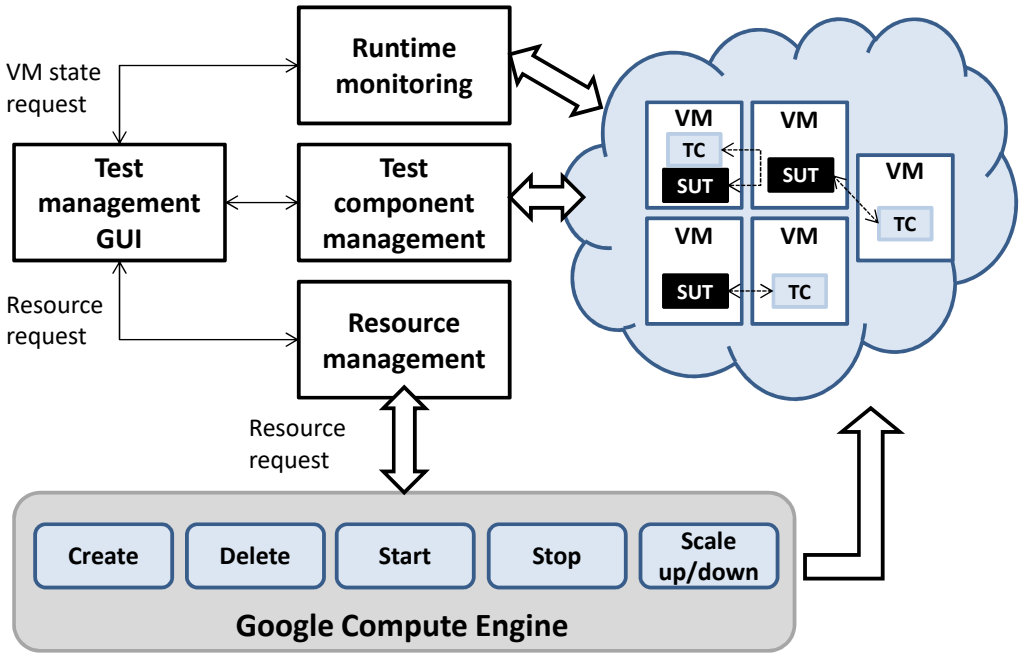


Figure 12.1: Test Execution Platform Overview.

Also, the proposed approach in (175) defines an automated test support as-a-service in order to enhance the self testing process. Similar to our proposal, this test harness is used for monitoring and validating dynamic adaptations. To avoid interference risk between test and business processes at runtime, it uses copies of services under test. Conversely, we adopt in our context testable services by using the Built-In Test (BIT) technique (179).

It should be noted that several commercial tools have been proposed to handle cloud testing, as well. For example, IBM provides its Infrastructure Optimization Services â“ IBM Smart Business Test Cloud in which on-demand secure, dynamic and scalable virtual test server resources are managed in a private test environment (180). SOASTA is another platform, called CloudTest, offering load testing capabilities from development to production (181).

12.4 Proposed Approach

The proposed approach is built based on TaaS concepts. Fig. 12.1 outlines an overview of its different constituents.

- **Test management GUI:** This component is charged with managing the overall testing process: dynamic allocation of test components to the appropriate VMs, start up of test component execution and computation of the final verdict. Moreover, it is responsible for querying the runtime monitoring component for information about the usage of resources in running VMs.

- **Resource management:** This component enables flexibility and elasticity during the testing process. If there is no adequate VM to handle the execution of a test component, a new VM can be created and started automatically. Moreover, it is possible to scale up or scale down an existing VM. The unused one can be released, as well.

- **Test component management:** This component offers services for creating/deleting test components and starting/stopping their execution. A test component is an entity that interacts with the SUT to execute the available test cases (i.e., a set of input values and expected results) and to observe its response related to this excitation. Its main role consists of stimulating the SUT with the input values, comparing the obtained output values to the expected results and generating the final verdict that can be pass, fail or inconclusive.

  Runtime monitoring: This component gives the status of each VM in terms of computing resources (such as CPU, memory, storage).

12.5 eHealth case study

With the aim of illustrating the usefulness of the proposed scalable test platform to ensure the trustworthiness and the correctness of critical systems, we adopt the TRMCS case study once again.

12.5.1 Implementation and deployment of TRMCS System

We used Apache Tomcat 7 as a Web server, Axis as a SOAP engine, Java 1.8 as a programming language, and MySQL 5 as a database management system. The obtained Web services are then deployed as depicted in Table 12.1 on a distributed environment composed of several standard virtual machines. Each one is characterized with 1 virtual CPU, 3.75 GB of memory and 10 GB of hard disk.

Table 12.1: Deployment of the TRMCS application.

Machine name	Machine type	SUT
instance1	n1-standard-1	Storage Service Analysis Service Alerting Service
instance2	n1-standard-1	Patient Service
instance3	n1-standard-1	Doctor Service
instance4	n1-standard-1	Sensors

12.5.2 Runtime testing

In order to perform runtime testing on the running TRMCS system without side effects (i.e., interference between test processes and business processes), we assume that all TRMCS services are equipped with a Built-In Test (BIT) interface (34). In fact, this technique consists in offering services with the facility to test themselves and their ability to be tested by their execution environment. To do so, they are equipped with a test interface and they are called testable services. The operations provided by each test interface ensure that the test data and business data are not mixed during the runtime testing process.

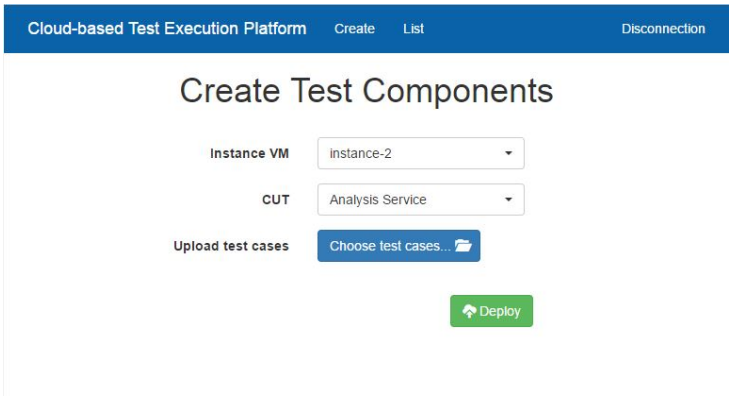


Figure 12.2: Screenshot of Test component creation and assignement GUI.

In the following, we highlight the usefulness of the proposed Cloud-based Test Execution Platform in order to perform runtime tests on the running TRMCS System. First of all, our platform is used to create Test Components and to deploy them in the adequate VM instance. To do so, the Graphical User Interface (GUI) outlined in Fig. 12.2 is Recall that test components assignment to VM instances is done in response to the current VM status. If there is a shortage of computing resources, a new VM instance can be created to hold the execution of the considered test component.

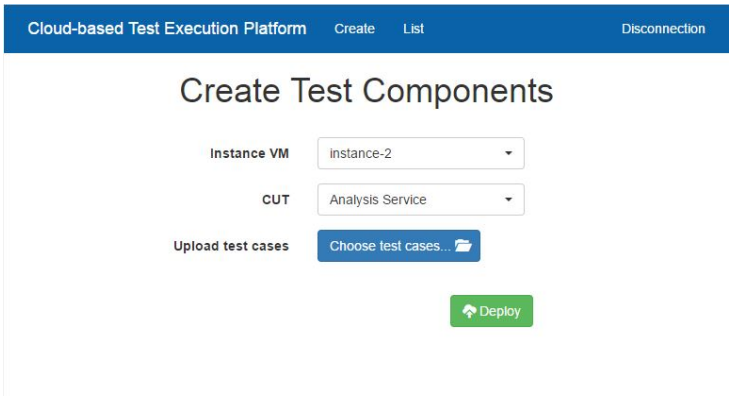


Figure 12.3: Screenshot of VM instance creation.

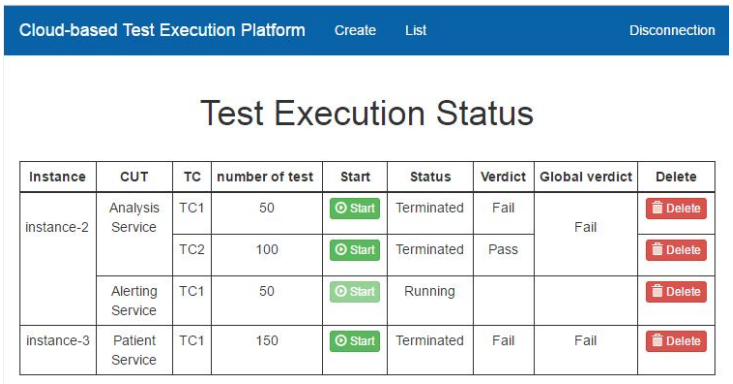


Figure 12.4: Screenshot of Test Execution GUI.

12.6 Summary

In this chapter we proposed a scalable test execution platform providing testing facilities as a service. Indeed, we defined not only a TaaS but also cloud services to monitor and manage automatically computing resources through creating/deleting/scaling up and down VMs. This platform allowed testers to set up the testing environment, assign test cases to the appropriate VMs, automatically execute them and collect and display test results. A proof of concept prototype was built and illustrated via a case study in the domain of eHealth.

As future work, we aim to extend our platform with test generation capabilities in the context of dynamically adaptable and distributed systems. The key idea here is to provide a model-based test generation as a service which can be executed after each dynamic behavioral adaptation. With the aim of reducing the cost of test generation, we investigate the use of Probabilistic Timed Automata (PTAs) as behavioral models from which tests will be generated (182). In this case, the obtained runtime tests cover essentially the most predictable behaviors. It will be also interesting to investigate runtime testing of Internet of Objects (IOT) applications and how our proposed platform may give an efficient solution to validate software services in an IoT context.

CHAPTER 13

General Conclusion

The present chapter concludes this dissertation, summarizes the presented contributions and proposes some future research directions to explore.

13.1 Summary

In this dissertation we reported on our main research contributions in the field of Model-Based Testing of Dynamic and Distributed Real-Time Systems, performed during the last ten years.

Our first contribution is related to testing techniques for distributed and dynamically adaptable systems. At this level, we proposed a standard-based test execution platform which offers a platform-independent test system for isolating and executing runtime tests. This platform explores the TTCN3 standard and considers both structural and behavioral adaptations. In addition, it has a test isolation layer that reduces the risk of interference between testing processes and business processes. Moreover, our platform computes a minimal subset of test cases and efficiently distributes them among the execution nodes. In addition, the proposed techniques were validated on two case studies, one in the healthcare domain and the other one in the fleet management domain.

In our second contribution, we proposed a model-based framework to combine Load and Functional Tests. For thi purpose, we used the model of extended timed automata with inputs/ouputs and shared integer variables. At this level, different modelling techniques illustrating some methodological aspects were introduced. Moreover, we examined BPEL compositions behaviors under various load conditions using the proposed framework. We also proposed a tax-

onomy of the detected problems and we illustrated how test verdicts are assigned. In addition, our approach was validated by applying our tool to a Travel Agency case study and several mutants of the corresponding BPEL process were considered.

In our third contribution, we proposed a set of formal techniques for the determinization and off-line test selection for timed automata with inputs and outputs. In this context, we proposed a game-based approach between two players for the determinization of a given timed automaton and some fixed resources. Furthermore, we introduced a complete formalization for the automatic off-line generation of test cases from non-deterministic timed automata with inputs and outputs. We also proposed a selection technique of test cases with expressive test purposes. Our method for generating test cases uses a symbolic co-reachability analysis of the observable behaviors of the specification guided by the test purpose defined as a special timed automaton.

Finally we reported on two ongoing works: (1) In the first one, we are interested in establishing a model-based approach for security testing of Internet of Things (IoT) applications; and (2) The goal of the second one is to provide a scalable test execution platform providing testing facilities as a cloud service.

13.2 Future Works

Many possible extensions for our work are possible. Next we list some possible directions to investigate in the future.

- **Meta-heuristic techniques for the constrained test placement problem:** The major problem that we faced while applying RTF4ADS on large-scale environments comes from the constrained test placement module. In fact, this module requires a long time to compute an exact optimal solution fitting the resource and connectivity constraints. Therefore, we intend to use the *Tabu Search* (TS) meta-heuristic as a resolution algorithm and performing a parallel exploration of the solution domain.

- **Extension of the distributed TTCN-3 Test System:** The current version of RTF4ADS focuses only on distributing TTCN-3 test cases. Each one is managed by a *Main Test Component* (MTC) and may create several Parallel Test Components (PTC) in order to execute integration tests. To gain more performance and to alleviate the test workload on the execution environment, we should also distribute PTC Components over the execution nodes in order to avoid the communication overhead introduced by the centralized execution architecture (67).

- **Runtime testing of autonomous systems:** We intend to enhance our test framework in order to support autonomous systems which are able generate emergent behaviors in response to changing environmental conditions. To do so, we should include our test system into *Monitor-Analyze-Plan-Execute* (MAPE-K) loops with the purpose of automating not only the adaptation process but also the runtime testing process.

- **Test generation based on probabilistic model-checking:** The key idea here is to apply runtime testing before the occurrence of dynamic proactive adaptations which consist in making predictions of how the environment or the system is going to evolve in the near future. To do so, tests have to be generated from behavioral models that are augmented with probabilities to describe the unpredictable system's behavior. Formalisms like *Probabilistic Timed Automata* can be used to specify the system behavior.

- **Distributed and resource-aware load testing of WS-BPEL compositions:** Recognizing problems under load is a challenging and time-consuming activity due to the large amount of generated data and the long running time of load tests. For this reason, we intend to extend our previous approach dealing with functional and load testing of BPEL compositions by distribution and resource awareness capabilities. Indeed, supporting test distribution over the network may alleviate considerably the test workload at runtime, especially when the SUT is running on a cluster of BPEL servers.

- **Developping heuristics to determinize timed automata:** The determinization of timed automata is a complex problem and our proposed algorithms run in time doubly exponential in the size of the input. Given the difficulty of the problem, it would be of interest to develop some heuristics. For instance, the resources and other features of our algorithms could be optimized online. During the on-the-fly construction of the game while searching for a winning strategy, resource clocks could be added if necessary, or the precision of the guards and relations could be increased.

- **Combining coverage with on-line test execution for real-time systems:** The topic of coverage needs to be studied in more depth in a real-time context. In particular, combining coverage with on-line test execution is another aspect that seems to be little studied. The problem is related to choosing online tester outputs and output times. Many heuristics can be applied to resolve such choices, but an additional problem is how to manage these choices across the execution of the entire test suite, using some appropriate book-keeping techniques.

13.3 List of Publications

Journals

- Hamilton Wilfried Yves Adoni, Tarik Nahhal, Moez Krichen, Brahim Aghezzaf, Abdeltif Elbyed. A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems. In *Journal of Distributed and Parallel Databases (2019).* (183)

- Moez Krichen. Improving Formal Verification and Testing Techniques for Internet of Things and Smart Cities. *Journal of Mobile Networks and Applications, MONET (2019).* (184) [French version available (185)]

- Mariam Lahami, Moez Krichen, Roobaea Alroobaea. TEPaaS: test execution platform as-a-service applied in the context of e-health. In *International Journal of Autonomous and Adaptive Communications Systems, IJAACS 12(3): 264-283 (2019).* (186)

- Moez Krichen, Afef Jmal Maâlej, Mariam Lahami. A model-based approach to combine conformance and load tests: an eHealth case study. In *International Journal of Critical Computer-Based Systems, IJCCBS 8(3/4): 282-310 (2018).* (187)

- Mariam Lahami, Moez Krichen. Safe and Efficient Runtime Testing Framework Applied in Dynamic and Distributed Systems. In *Science of Computer Programming Journal. 122: 1-28 (2016).* (164)

- Afef Jmal Maâlej, Moez Krichen, Mohamed Jmaiel. A Comparative Evaluation of State-of-the-Art Load and Stress Testing Approaches. In *International Journal of Computer Applications in Technology IJCAT 51(4): 283-293 (2015).* (8)

- Mariam Lahami, Moez Krichen, Mohamed Jmaiel. Runtime Testing Approach of Structural Adaptations for Dynamic and Distributed Systems. In *International Journal of Computer Applications in Technology. IJCAT 51(4): 259-272 (2015).* (188)

- Afef Jmal Maâlej, Moez Krichen. Study on the Limitations of WS-BPEL Compositions Under Load Conditions. In *The Computer Journal (2015) 58 (3): 385-402* (189)

- Nathalie Bertrand, Amélie Stainer, Thierry Jéron, Moez Krichen. A game approach to determinize timed automata. In *Formal Methods in System Design 46(1): 42-80 (2015).* (190)

- Nathalie Bertrand, Thierry Jéron, Amélie Stainer, Moez Krichen: Off-line test selection with test purposes for non-deterministic timed automata. In *Logical Methods in Computer Science 8(4) (2012)*. (191)

- Moez Krichen: A formal framework for black-box conformance testing of distributed real-time systems. In *International Journal of Critical Computer-Based Systems, IJCCBS 3(1/2): 26-43. 2012*. (192) [Arabic version available (193)]

- Mariam Lahami, Moez Krichen and Mohamed Jmaiel. A distributed Test Architecture For Adaptable and Distributed Real-Time Systems.In *the Journal of New technologies of Information. 2012*. (194)

- Moez Krichen and Stavros Tripakis. Conformance Testing for Real-Time Systems. In *Formal Methods in System Design*, 34(3): 238–304. Elsevier, 2009. (2)

- Saddek Bensalem, Moez Krichen, Lotfi Majdoub, Riadh Robbana, and Stavros Tripakis. A simplified approach for testing real-time systems based on action refinement. In *ISoLA*, volume RNTI-SM-1 of *Revue des Nouvelles Technologies de l'Information*, pages 191–202. Cépaduès-Éditions, 2007. (195)

- Patricia Bouyer, Fabrice Chevalier, Moez Krichen, et Stavros Tripakis. Observation partielle des systèmes temporisés. Dans le *Journal Européen des Systèmes Automatisés, Actes du 5ème Colloque sur la Modélisation des Systèmes Réactifs, MSR 2005, Autrans, France, 5-7 Octobre 2005*, pages 381–393. Hermès, 2005. Papier invité. (196)

- Saddek Bensalem, Marius Bozga, Moez Krichen, and Stavros Tripakis. Testing conformance of real-time applications by automatic generation of observers. *Electronic Notes in Theoretical Computer Science*, 113:23–43. Elsevier, 2005. (197)

Book Chapters

- Moez Krichen and Mariam Lahami. Towards a Runtime Testing Framework for Dynamically Adaptable Internet of Things Networks in Smart Cities. In *Mehmood R., See S., Katib I., Chlamtac I. (eds) Smart Infrastructure and Applications. EAI/Springer Innovations in Communication and Computing.* EAI/Springer Innovations in Communication and Computing, 2020. (198)

- Moez Krichen, Mariam Lahami, Omar Cheikhrouhou, Roobaea Alroobaea and Afef Jmal Maâlej. Security Testing of Internet of Things for Smart City Applications: A Formal Approach. In *Mehmood R., See S., Katib I., Chlamtac I. (eds) Smart Infrastructure and Ap-*

plications. EAI/Springer Innovations in Communication and Computing. EAI/Springer Innovations in Communication and Computing, 2020. (199)

□ Moez Krichen. State identification. In *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*, pages 35–67. Springer, 2004. (200)

Conferences

□ Moez Krichen. Improving and Optimizing Verification and Testing Techniques for Distributed Information Systems. In *Proceedings of the 21st International Conference on Enterprise Information Systems (ICEIS 2019), Heraklion, Crete, Greece, May 3-5. (Revised Selected Papers)* Springer, Cham, 2019. (184)

□ Moez Krichen. Testing Real-Time Systems using Determinization Techniques for Automata over Timed Domains. In the *Proceedings of the 16th International Colloquium on Theoretical Aspects of Computing (ICTAC 2019), October 31 - November 4, 2019, Springer 2019.* (201)

□ Moez Krichen, Roobaea Alroobaea. Towards Optimizing the Placement of Security Testing Components for Internet of Things Architecture. In The *Proceedings of the 16th ACS/IEEE International Conference on Computer Systems and Applications, (AICCSA 2019), Abu Dhabi, UAE, November 3-7, 2019, IEEE 2019.*

□ Moez Krichen, Wilfried Yves Hamilton Adoni, Tarik Nahhal. Some Placement Techniques of Test Components for Smart Cities and Internet of Things Inspired by Fog Computing Approaches. To Appear In The *Proceedings of the The First International Conference on Smart Information & Communication Technologies, (SmartICT 2019), Saïdia, Morocco, September 26-28, 2019. Springer 2019.*

□ Wilfried Yves Hamilton Adoni, Moez Krichen, Tarik Nahhal. Multi-path Coverage of all Final States for Model-Based Testing using Spark In-memory Design. To Appear In The *Proceedings of the The First International Conference on Smart Information & Communication Technologies, (SmartICT 2019), Saïdia, Morocco, September 26-28, 2019. Springer 2019.*

□ Moez Krichen, Roobaea Alroobaea. A New Model-based Framework for Testing Security of IoT Systems in Smart Cities using Attack Trees and Price Timed Automata. In The

Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, (ENASE 2019), Heraklion, Crete, Greece, May 4-5, 2019, pages 570-577, SciTePress 2019. (202)

- Moez Krichen, Roobaea Alroobaea, Mariam Lahami. Towards a Runtime Standard-based Testing Framework for Dynamic Distributed Information Systems. In *Proceedings of the 21st International Conference on Enterprise Information Systems (ICEIS 2019), Heraklion, Crete, Greece, May 3-5, 2019, pages 121-129, Volume 2. SciTePress 2019.* (203)

- Mariam Lahami, Moez Krichen, Roobaea Alroobaea. Towards a Test Execution Platform As-A-Service: Application in the E-Health Domain. In *Proceedings of the 2nd International Conference on Control, Automation and Diagnosis (ICCAD 2018), Marrakech, Morocco, March 19-21, 2018. IEEE, 2018.* (204)

- Moez Krichen, Afef Jmal Maâlej, Mariam Lahami, Mohamed Jmaiel. A Resource-Aware Model-Based Framework for Load Testing of WS-BPEL Compositions. In *Enterprise Information Systems - 20th International Conference, ICEIS 2018, Funchal, Madeira, Portugal, March 21-24, 2018, Revised Selected Papers, pages 130-157, LNBIP, volume 363. Springer, 2018.* (205)

- Afef Jmal Maâlej, Mariam Lahami, Moez Krichen, Mohamed Jmaiel. Distributed and Resource-Aware Load Testing of WS-BPEL Compositions. In *Proceedings of the 20th International Conference on Enterprise Information Systems (ICEIS 2018), Funchal, Madeira, Portugal, March 21-24, 2018, Volume 2. SciTePress 2018.* (206)

- Moez Krichen, Omar Cheikhrouhou, Mariam Lahami, Roobaea Alroobaea, Afef Jmal Maâlej. Towards a Model-Based Testing Framework for the Security of Internet of Things for Smart City Applications. In *Proceedings of the 1st EAI International Conference on Smart Societies Infrastructure, Technologies, and Applications (SCITA 2017), Jeddah, Saudi Arabia, November 27-29, 2017. Springer, 2017.* (207)

- Afef Jmal Maâlej, Moez Krichen. WSCLim: A Tool for Model-Based Testing of WS-BPEL Compositions Under Load Conditions. In *Proccedings of the 11th International Conference on Tests and Proofs (TAP 2017), pages 139-151, Marburg, Germany, July 19-20, 2017. Springer, 2017.* (208)

- Mariam Lahami, Moez Krichen, Hajer Barhoumi, Mohamed Jmaiel. Selective Test Generation Approach for Testing Dynamic Behavioral Adaptations. In *Proccedings of the 27th IFIP International Conference of Testing Software and Systems (ICTSS 2015), pages*

224-239, Sharjah and Dubai, United Arab Emirates, November 23-25, 2015. Springer, 2015. (209)

□ Mariam Lahami, Moez Krichen. Test Isolation Policy for Safe Runtime Validation of Evolvable Software Systems. *In Proceedings of the 22nd IEEE International Conference on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2013), pages 377-382, Hammamet, Tunisia, June 17-20, 2013, IEEE Computer Society.* (209)

□ Afef Jmal Maâlej, Manel Hamza, Moez Krichen. WSCLT: A Tool for WS-BPEL Compositions Load Testing. *In Proceedings of the 22nd IEEE International Conference on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2013), pages 272-277, Hammamet, Tunisia, June 17-20, 2013, IEEE Computer Society.* (210)

□ Mariam Lahami, Fairouz Fakhfakh, Moez Krichen, Mohamed Jmaiel. Towards a TTCN-3 Test System for Runtime Testing of Adaptable and Distributed Systems. In *Proccedings of the 23rd IFIP International Conference of Testing Software and Systems, ICTSS 2012, Aalborg, Denmark, November 19 - 21, 2012. Springer, 2012.* (84)

□ Mariam Lahami, Moez Krichen, Mariam Bouchakwa, Mohamed Jmaiel. Using Knapsack Problem Model to Design a Resource Aware Test Architecture for Adaptable and Distributed Systems. In Proceedings of the 23rd IFIP International Conference of Testing Software and Systems, ICTSS 2012, Aalborg, Denmark, November 19 - 21, 2012. Springer, 2012 (83)

□ Mariam Lahami, Moez Krichen, Mohamed Jmaiel. A distributed Test Architecture For Adaptable and Distributed Real-Time Systems. In *Proceedings of 'ConfÃ©rence sur les Architectures Logicielles',CAL 2011, Lille, France, June 2011.* (194)

□ Nathalie Bertrand, Amélie Stainer, Thierry Jéron, Moez Krichen. A Game Approach to Determinize Timed Automata. In *Proceedings of the 14th International Conference on Foundations of Software Science and Computation Structures, FoSSaCS'11, SaarbrÃ¼cken, Germany, April 2011. LNCS 6604, pages 245-259. Springer,2011.* (157)

□ Nathalie Bertrand, Amélie Stainer, Thierry Jéron, Moez Krichen. Off-line Test Selection with Test Purposes for Non-Deterministic Timed Automata. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11), SaarbrÃ¼cken, Germany, April 2011. LNCS 6605, pages 96-111. Springer, 2011.* (211)

- Moez Krichen. 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, December 14-17, 2010. LNCS 6490, pages 139-142. Springer, 2010.* (212)

- Mariam Lahami, Moez Krichen, Akram Idani, Mohamed Jmaiel. A generic process to build reliable distributed software components from early to late stages of software development. In *Sixth International Conference on Computer Engineering and Systems, ICCES 2010, Cairo, Egypt, November 30 - December 2, 2010, Proceedings.* (213)

- Moez Krichen, Monika Solanki. Automatic Generation of Real-Time Observers for Monitoring Web Services. In *Second International Conference on Web and Information Technologies, ICWIT 2009, Kerkennah Islands, Sfax, Tunisia, June 12 - 14, 2009, Proceedings.* (214)

- Matthieu Gallien, Fahmi Gargouri, Imen Kahloul, Moez Krichen, Thanh Hung Nguyen, Saddek Bensalem, Félix Ingrand. D'une approche modulaire à une approche orientée composant pour le développement de systèmes autonomes : défis et principes. In *3rd National Conference on Control Architectures of Robots, CAR 2008, Bourges, France, May 29 - 30, 2008, Proceedings.* Invited paper. (215)

- Saddek Bensalem, Marius Bozga, Matthieu Gallien, Félix Ingrand, Moez Krichen, Stavros Tripakis. Automatic generation of observers for the dala robot with ttg. In *1st Mediterranean Conference on Intelligent Systems and Automation, CISA 2008, Annaba, Algeria, June 30 - July 02, 2008, Proceedings, volume 1019 of American Institute of Physics*, pages 487–492. AIP, 2008. (216)

- Saddek Bensalem, Moez Krichen, Stavros Tripakis. Generating Analog-Clock Real-Time Testers Using Action Refinement Techniques. Dans les *Actes de la Conférence Internationale sur les Relations, Ordres et Graphes: Interaction avec l'Informatique, ROGICS 2008, Mahdia, Tunisie, 12-17 Mai 2008.*

- Moez Krichen, Stavros Tripakis. Interesting properties of the real-time conformance relation tioco. In *Theoretical Aspects of Computing - ICTAC 2006, Third International Colloquium, Tunis, Tunisia, November 20-24, 2006, Proceedings*, volume 4281 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2006. (217)

- Moez Krichen, Stavros Tripakis. An expressive and implementable formal framework for testing real-time systems. In *Testing of Communicating Systems, 17th IFIP TC6/WG*

6.1 International Conference, TestCom 2005, Montreal, Canada, May 31 - June 2, 2005, Proceedings, volume 3502 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2005. (218)

□ Moez Krichen, Stavros Tripakis. State identification problems for timed automata. In *Testing of Communicating Systems, 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005, Montreal, Canada, May 31 - June 2, 2005, Proceedings*, volume 3502 of *Lecture Notes in Computer Science*, pages 175–191. Springer, 2005. (219)

□ Moez Krichen, Stavros Tripakis. Real-time testing with timed automata testers and coverage criteria. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, volume 3253 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2004. (220)

Workshops

□ Afef Jmal Maâlej, Moez Krichen, Mohamed Jmaiel. A Model Based Approach to Combine Load and Functional Tests for Service Oriented Architectures. In *Proccedings of the 10th Workshop on Verification and Evaluation of Computer and Communication System (VECoS 2016), Tunis, Tunisia, October 6-7, 2016. CEUR-WS.org 2016.* (221)

□ Mariam Lahami, Moez Krichen, Mohamed Jmaiel. Runtime Testing Framework for Improving Quality in Dynamic Service-based Systems. *In Proceedings of the 2nd International Workshop on Quality Assurance for Service-Based Applications (QASBA 2013) in conjunction with the International Symposium in Software Testing and Analysis (ISSTA 2013), pages 17-24 , Lugano, Switzerland, July 2013.* ACM. (222)

□ Afef Jmal Maâlej, Zeineb Ben Makhlouf, Moez Krichen, Mohamed Jmaiel. Conformance Testing for Quality Assurance of Clustering Architectures. *In Proceedings of the 2nd International Workshop on Quality Assurance for Service-Based Applications (QASBA 2013) in conjunction with the International Symposium in Software Testing and Analysis (ISSTA 2013), pages 9-16, Lugano, Switzerland, July 2013.* ACM. (223)

□ Afef Jmal Maâlej, Manel Hamza, Moez Krichen, Mohamed Jmaiel. Automated Significant Load Testing for WS-BPEL Compositions. *In Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2013), pages 144-153, Luxembourg, March 18-22, 2013, IEEE Computer Society.* (130)

- Afef Jmal Maâlej, Moez Krichen, Mohamed Jmaiel. WSCCT: A Tool for WS-BPEL Compositions Conformance Testing. *In Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 2013), pages 1055-1061, Coimbra, Portugal, March 18-22, 2013, ACM.* (224)

- Afef Jmal Maâlej, Moez Krichen, Mohamed Jmaiel. Model-based Conformance Testing of WS-BPEL Compositions. *In Proceedings of the IEEE 36th International Conference on Computer Software and Applications Workshops, COMPSAC 2012, pages 452-457, Izmir, Turkey, July 16-20, 2012.* IEEE Computer Society. (225)

- Saddek Bensalem, Moez Krichen, Stavros Tripakis. State Identification Problems for Input/Output Transition Systems. In *9th International Workshop on Discrete Event Systems, WODES 2008, Göteborg, Sweden, May 28 - 30 2008 , Proceedings,* pages 225–230. IEEE, 2008. (226)

- Saddek Bensalem, Moez Krichen, Lotfi Majdoub, Riadh Robbana, Stavros Tripakis. Test Generation for Duration Systems. In *First International Workshop on Verification and Evaluation of Computer and Communication Systems, VECoS 2007, Algiers, Algeria, 5 - 6 May 2007, Proceedings.* British Computer Society - BCS, 2007. (227)

- Moez Krichen, Stavros Tripakis. State-identification problems for finite-state transducers. In *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, volume 4262 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2006. (228)

- Saddek Bensalem, Marius Bozga, Moez Krichen, Stavros Tripakis. Testing Conformance of Real-Time Applications: Case of Planetary Rover Controller. In *Verification and Validation of Model-Based Planning and Scheduling Systems, VVPS 2005, Monterey, California, June 6-7, Proceedings.* Invited paper. (229)

- Moez Krichen, Stavros Tripakis. Black-box conformance testing for real-time systems. In *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 109–126. Springer, 2004. (230)

Bibliography

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., 2006.

[2] M. Krichen and S. Tripakis, "Conformance testing for real-time systems," *Formal Methods in System Design*, vol. 34, no. 3, pp. 238–304, 2009.

[3] C. Baier, N. Bertrand, P. Bouyer, and T. Brihaye, "When are timed automata determinizable?" in *ICALP'09*, ser. LNCS, vol. 5556, 2009, pp. 43–54.

[4] J. Kienzle, N. Guelfi, and S. Mustafiz, *Transactions on Aspect-Oriented Software Development VII: A Common Case Study for Aspect-Oriented Modeling.* Springer Berlin Heidelberg, 2010, ch. Crisis Management Systems: A Case Study for Aspect-Oriented Modeling, pp. 1–22.

[5] I.-Y. Chen and C.-H. Tsai, "Pervasive Digital Monitoring and Transmission of Pre-Care Patient Biostatics with an OSGi, MOM and SOA Based Remote Health Care System," in *Proceeding of the 6th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom'06)*, 2008, pp. 704–709.

[6] U. Varshney, "Pervasive Healthcare and Wireless Health Monitoring," *Mobile Networks and Applications*, vol. 12, no. 2-3, pp. 113–127, 2007.

[7] S. T. S. Thong, C. T. Han, and T. A. Rahman, "Intelligent Fleet Management System with Concurrent GPS GSM Real-Time Positioning Technology," in *Proceeding of the 7th International Conference on Intelligent Transport Systems Telecommunications (ITST'07)*, 2007, pp. 1–6.

[8] A. J. Maâlej, M. Krichen, and M. Jmaïel, "A comparative evaluation of state-of-the-art load and stress testing approaches," *Int. J. Comput. Appl. Technol.*, vol. 51, no. 4, pp. 283–293, Jul. 2015.

[9] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[10] M. Krichen, "Model-based testing for real-time systems," Ph.D. dissertation, PhD thesis, University of Joseph Fourier (December 2007), 2007.

[11] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[12] M. Lahami, "Runtime testing of dynamically adaptable and distributed component based Systems," Theses, Ecole Nationale d'Ingénieurs de Sfax, Apr. 2017. [Online]. Available: https://hal.archives-ouvertes.fr/tel-02469999

[13] J. Kramer and J. Magee, "Dynamic Configuration for Distributed Systems," *IEEE Transactions on Software Engineering (TSE)*, vol. 11, no. 4, pp. 424–436, 1985.

[14] A. Ketfi, N. Belkhatir, and P. yves Cunin, "Dynamic Updating of Component-Based Applications," in *Proceeding of the International Conference on Software Engineering Research and Practice (SERP'02)*, 2002.

[15] *OSGi service gateway specification, Release 4* , Open Services Gateway Initiative, 2005.

[16] G. Tamura, N. Villegas, H. Müller, J. Sousa, B. Becker, G. Karsai, S. Mankovskii, M. Pezzï¿½, W. Schï¿½fer, L. Tahvildari, and K. Wong, "Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems," in *Software Engineering for Self-Adaptive Systems II*, 2013, pp. 108–132.

[17] B. Cheng, K. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. Müller, P. Pelliccione, A. Perini, N. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. Villegas, "Using Models at Runtime to Address Assurance for Self-Adaptive Systems," in *Models@run.time*, 2014, pp. 101–136.

[18] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, "Models@Runtime to Support Dynamic Adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[19] P. Inverardi and M. Mori, "Model Checking Requirements at Run-time in Adaptive Systems," in *Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems (ASAS'11)*, 2011, pp. 5–9.

[20] P. Stocks and D. Carrington, "A Framework for Specification-Based Testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 22, no. 11, pp. 777–793, 1996.

[21] L. Liu, H. Miao, and X. Zhan, "A Framework for Specification-Based Class Testing," in *Proceeding of the 8th International Conference on Engineering of Complex Computer Systems (ICECCS'02)*, 2002, pp. 153–162.

[22] S. K. Swain and D. P. Mohapatra, "Test Case Generation from Behavioral UML Models," *International Journal of Computer Applications*, vol. 6, no. 8, pp. 5–11, 2010.

[23] A. Calvagna and A. Gargantini, "A Logic-based Approach to Combinatorial Testing With Constraints," in *Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08)*, 2008, pp. 66–83.

[24] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test Generation via Dynamic Symbolic Execution for Mutation Testing," in *Procceeding of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, 2010, pp. 1–10.

[25] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized Symbolic Execution for Model Checking and Testing," in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, 2003, pp. 553–568.

[26] ETSI, "Methods for Testing and Specification (MTS), The Testing and Test Control Notation version 3, Part 1: TTCN-3 Core Language," 2005.

[27] A. Khoumsi, "Testing Distributed Real Time Systems Using a Distributed Test Architecture," in *Proceeding of the IEEE Symposium on Computers and Communications (ISCC'01)*, 2001, pp. 648–654.

[28] S. Siddiquee and A. En-Nouaary, "Two Architectures for Testing Distributed Real-Time Systems," in *Proceeding of the 2nd Information and Communication Technologies (ICTTA'06)*, vol. 2, 2006, pp. 3388–3393.

[29] A. Tarhini and H. Fouchal, "Conformance Testing of Real-Time Component Based Systems," in *Proceeding of the International School and Symposium on Advanced Distributed Systems (ISSADS'05)*, 2005, pp. 167–181.

[30] A. Khoumsi, "Testing Distributed Real-Time reactive Systems Using a centralized Test Architecture," in *Proceeding of the North Atlantic Test Workshop (NATW)*, 2001, pp. 648–654.

[31] M. J. Harrold, "Testing: a roadmap," in *Proceedings of the 16th IEEE Conference on The Future of Software Engineering (ICSE'00)*, 2000, pp. 61–72.

[32] G. Rothermel and M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 173–210, 1997.

[33] H. Leung and L. White, "Insights into regression testing [software testing]," in *Proceedings of the International Conference on Software Maintenance (ICSM'89)*, 1989, pp. 60–69.

[34] D. Brenner, C. Atkinson, R. Malaka, M. Merdes, B. Paech, and D. Suliman, "Reducing Verification Effort in Component-based Software Engineering Through Built-In Testing," *Information Systems Frontiers*, vol. 9, no. 2-3, pp. 151–162, 2007.

[35] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," 1990.

[36] A. González, E. Piel, and H.-G. Gross, "A Model for the Measurement of the Runtime Testability of Component-Based Systems," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW'09)*, 2009, pp. 19–28.

[37] Y. Wang, G. King, D. Patel, S. Patel, and A. Dorling, "On Coping With Real-time Software Dynamic Inconsistency by Built-In Tests," *Annals of Software Engineering*, vol. 7, no. 1-4, pp. 283–296, 1999.

[38] J. Vincent, G. King, P. Lay, and J. Kinghorn, "Principles of Built-In-Test for Run-Time-Testability in Component-Based Software Systems," *Software Quality Control*, vol. 10, no. 2, pp. 115–133, 2002.

[39] É. Piel, A. González-Sanchez, and H.-G. Groß, "Automating Integration Testing of Large-Scale Publish/Subscribe Systems," in *Principles and Applications of Distributed Event-Based Systems*, 2010, pp. 140–163.

[40] L. Chu, K. Shen, H. Tang, T. Yang, and J. Zhou, "Dependency Isolation for Thread-based Multi-tier Internet Services," in *Proceeding of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, 2005, pp. 796–806.

[41] I. Granja and M. Jino, "Techniques for Regression Testing: Selecting Test Case Sets Tailored to Possibly Modified Functionalities," in *Proceedings of the 3rd European Conference on Software Maintenance and Reengineering (CSMR'99)*, 1999, pp. 2–22.

[42] A. Beszedes, T. Gergely, L. Schrettner, J. Jasz, L. Lango, and T. Gyimothy, "Code Coverage-Based Regression Test Selection and Prioritization in WebKit," in *Proceeding of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*, 2012, pp. 46–55.

[43] B. Korel and A. M. Al-Yami, "Automated Regression Test Generation," *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 2, pp. 143–152, 1998.

[44] B. Korel, L. Tahat, and B. Vaysburg, "Model Based Regression Test Reduction Using Dependence Analysis," in *Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM'02)*, 2002, pp. 214–223.

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

[46] Y. Chen, R. L. Probert, and H. Ural, "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)*, 2007, pp. 54–62.

[47] L. C. Briand, Y. Labiche, and S. He, "Automating Regression Test Selection Based on UML Designs," *Information & Software Technology*, vol. 51, no. 1, pp. 16–30, 2009.

[48] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon, "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)*, 2011, pp. 125–134.

[49] M. J. Harrold, "Architecture-Based Regression Testing of Evolving Systems," in *Proceeding of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA'98)*, 1998, pp. 73–77.

[50] H. Muccini, M. S. Dias, and D. J. Richardson, "Software Architecture-Based Regression Testing," *Journal of Systems and Software*, vol. 79, no. 10, pp. 1379–1396, 2006.

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

[52] M. Merdes, R. Malaka, D. Suliman, B. Paech, D. Brenner, and C. Atkinson, "Ubiquitous RATs: How Resource-Aware Run-Time Tests Can Improve Ubiquitous Software Systems,"

in *Proceedings of the 6th International Workshop on Software Engineering and Middleware (SEM'06)*, 2006, pp. 55–62.

[53] A. González-Sanchez, É. Piel, and H.-G. Gross, "Architecture Support for Runtime Integration and Verification of Component-based Systems of Systems," in *Proceeding of the Automated Software Engineering - Workshops, (ASE Workshops'08)*, 2008, pp. 41–48.

[54] C. Murphy, G. Kaiser, I. Vo, and M. Chu, "Quality Assurance of Software Applications Using the In Vivo Testing Approach," in *Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST'09)*, 2009, pp. 111–120.

[55] É. Piel and A. González-Sanchez, "Data-flow Integration Testing Adapted to Runtime Evolution in Component-Based Systems," in *Proceedings of the ESEC/FSE Workshop on Software Integration and Evolution@runtime*, 2009, pp. 3–10.

[56] D. Niebuhr and A. Rausch, "Guaranteeing Correctness of Component Bindings in Dynamic Adaptive Systems Based on Runtime Testing," in *Proceedings of the 4th International Workshop on Services Integration in Pervasive Environments (SIPE'09)*, 2009, pp. 7–12.

[57] M. Greiler, H.-G. Gross, and A. van Deursen, "Evaluation of Online Testing for Services: A Case Study," in *Proceeding of the 2nd International Workshop on Principles of Engineering Service-Oriented System*, 2010, pp. 36–42.

[58] T. M. King, A. A. Allen, R. Cruz, and P. J. Clarke, "Safe Runtime Validation of Behavioral Adaptations in Autonomic Software," in *Proceedings of the 8th International Conference on Autonomic and Trusted Computing (ATC'11)*, 2011, pp. 31–46.

[59] X. Bai, D. Xu, G. Dai, W.-T. Tsai, and Y. Chen, "Dynamic Reconfigurable Testing of Service-Oriented Architecture," in *Proceeding of the 31st Annual International Computer Software and Applications Conference (COMPSAC'07)*, 2007, pp. 368–378.

[60] J. Hielscher, R. Kazhamiakin, A. Metzger, and M. Pistore, "A Framework for Proactive Self-adaptation of Service-Based Applications Based on Online Testing," in *Proceedings of the 1st European Conference on Towards a Service-Based Internet (ServiceWave'08)*, 2008, pp. 122–133.

[61] A. E. Ramirez, B. Morales, and T. M. King, "A Self-Testing Autonomic Job Scheduler," in *Proceedings of the 46th Annual Southeast Regional Conference on XX (ACM-SE'08)*, 2008, pp. 304–309.

[62] X. Bai, G. Dai, D. Xu, and W.-T. Tsai, "A Multi-Agent Based Framework for Collaborative Testing on Web Services," in *Proceedings of the 4th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the 2nd International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*, 2006, pp. 205–210.

[63] M. Akour, A. Jaidev, and T. M. King, "Towards Change Propagating Test Models in Autonomic and Adaptive Systems," in *Proceedings of the 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS'11)*, 2011, pp. 89–96.

[64] E. M. Fredericks, B. DeVries, and B. H. C. Cheng, "Towards Run-time Adaptation of Test Cases for Self-adaptive Systems in the Face of Uncertainty," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*, 2014, pp. 17–26.

[65] D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes, and R. Malaka, "The MORABIT Approach to Runtime Component Testing," in *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC '06)*, 2006, pp. 171–176.

[66] S. Schulz and T. Vassiliou-Gioles, "Implementation of TTCN-3 Test Systems using the TRI," in *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems (TestCom'02)*, 2002, pp. 425–442.

[67] I. Schieferdecker and T. Vassiliou-Gioles, "Realizing Distributed TTCN-3 Test Systems With TCI," in *Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems (TestCom'03)*, 2003.

[68] G. Din, S. Tolea, and I. Schieferdecker, "Distributed Load Tests with TTCN-3," in *Proceedings of the 18th IFIP TC6/WG6.1 International Conference for Testing of Communicating Systems (TestCom'06)*, 2006, pp. 177–196.

[69] B. Stepien, L. Peyton, and P. Xiong, "Framework Testing of Web Applications Using TTCN-3," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 10, no. 4, pp. 371–381, 2008.

[70] Q. L. Ying Li, "Research on Web Application Software Load Test Using Technology of TTCN-3," *American Journal of Engineering and Technology Research*, vol. 11, pp. 3686–3690, 2011.

[71] I. Schieferdecker, G. Din, and D. Apostolidis, "Distributed Functional and Load Tests for Web Services," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, pp. 351–360, 2005.

[72] C. Rentea, I. Schieferdecker, and V. Cristea, "Ensuring Quality of Web Applications by Client-side Testing Using TTCN-3," in *Proceeding of the 21th IFIP International Conference on Testing of Communicating Systems joint with 9th International Workshop on Formal Approaches to Testing of Software (TestCom/Fates'09)*, 2009.

[73] J. C. Okika, A. P. Ravn, Z. Liu, and L. Siddalingaiah, "Developing a TTCN-3 Test Harness for Legacy Software," in *Proceedings of the International Workshop on Automation of Software Test*, 2006, pp. 104–110.

[74] D. A. Serbanescu, V. Molovata, G. Din, I. Schieferdecker, and I. Radusch, "Real-Time Testing with TTCN-3," in *Proceeding of the 20th IFIP International Conference on Testing of Communicating Systems joint with 8th International Workshop on Formal Approaches to Testing of Software (TestCom/Fates'08)*, 2008, pp. 283–301.

[75] P. H. Deussen, G. Din, and I. Schieferdecker, "A TTCN-3 Based Online Test and Validation Platform for Internet Services," in *Proceedings of the 6th International Symposium on Autonomous Decentralized Systems (ISADS'03)*, 2003.

[76] B. Li, Y. Zhou, Y. Wang, and J. Mo, "Matrix-based Component Dependence Representation and Its Applications in Software Quality Assurance," *ACM SIGPLAN Notices*, vol. 40, no. 11, pp. 29–36, 2005.

[77] S. Alhazbi and A. Jantan, "Dependencies Management in Dynamically Updateable Component-Based Systems," *Journal of Computer Science*, vol. 3, no. 7, pp. 499–505, 2007.

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

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

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

[81] G. Rothermel and M. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[82] K. Ghédira and B. Dubuisson, *Constraint Satisfaction Problems.* John Wiley & Sons, Inc., 2013, ch. Foundations of CSP, pp. 1–28.

[83] M. Lahami, M. Krichen, M. Bouchakwa, and M. Jmaïel, "Using Knapsack Problem Model to Design a Resource Aware Test Architecture for Adaptable and Distributed Systems," in *Proceedings of the 24th IFIP WG 6.1 International Conference Testing Software and Systems (ICTSS'12)*, 2012, pp. 103–118.

[84] M. Lahami, F. Fakhfakh, M. Krichen, and M. Jmaïel, "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)*, 2012, pp. 71–86.

[85] G. Behrmann, A. David, and K. Larsen, "A tutorial on uppaal," in *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, ser. LNCS, M. Bernardo and F. Corradini, Eds., vol. 3185. Springer Verlag, 2004, pp. 200–237.

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

[87] A. Hessel and P. Pettersson, "CO$\sqrt{}$ER A Real-Time Test Case Generation Tool," in *Proceeding of the 7th International Workshop on Formal Approaches to Testing of Software (FATES'07)*, 2007.

[88] A. Hessel, "Model-based test case generation for real-time systems," Ph.D. dissertation, Uppsala University, Sweden, 2007.

[89] M. Beyer, W. Dulz, and F. Zhen, "Automated TTCN-3 Test Case Generation by Means of UML Sequence Diagrams and Markov Chains," in *Proceeding of the 12th Asian Test Symposium (ATS'03)*, 2003, pp. 102–105.

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

[91] J. P. Ernits, A. Kull, K. Raiend, and J. Vain, "Generating TTCN-3 Test Cases from EFSM Models of Reactive Software Using Model Checking," in *Informatik 2006 - Informatik für Menschen, Band 2, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2.-6*, 2006, pp. 241–248.

[92] D. E. Vega, G. Din, and I. Schieferdecker, "Application of TTCN-3 Test Language to Testing Information Systems in eHealth Domain," in *Procceding of the International Conference on Multimedia Computing and Information Technology (MCIT'10)*, 2010, pp. 21–24.

[93] N. Katanić, T. Nenadić, S. Devsic, and L. Skorin-Kapov, "Automated Generation of TTCN-3 Test Scripts for SIP-based Calls," in *Proceedings of the 33rd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO'10)*, 2010, pp. 423–427.

[94] X. Zhao and W. Zheng, "Research and Application on MBT and TTCN-3 Based Automatic Testing Approach," in *Proceeding of the International Conference on Computer Application and System Modeling (ICCASM'10)*, vol. 1, 2010, pp. 481–485.

[95] J. Zander, Z. R. Dai, I. Schieferdecker, and G. Din, "From u2tp models to executable tests with ttcn-3 - an approach to model driven testing -," in *Proceedings of 17th IFIP TC6/WG 6.1 International Conference for Testing Communicating Systems (TestCom'05)*, 2005, pp. 289–303.

[96] T. V. Axel Rennoch, Claude Desroches and I. Schieferdecker, "TTCN-3 Quick Reference Card," 2016.

[97] T. Technologies, "TTthree - Compile TTCN-3 modules into test executables," http://www.testingtech.com/products/, 2008.

[98] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, "GraphML Progress Report," in *Proceeding of the International Symposium on Graph Drawing (GD'01)*, 2001, pp. 501–512.

[99] N. Jussien, G. Rochart, and X. Lorca, "Choco: an Open Source Java Constraint Programming Library," in *Proceeding of the Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2008, pp. 1–10.

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

[101] P. Inverardi, C. Mangano, F. Russo, and S. Balsamo, "Performance Evaluation of a Software Architecture: a Case Study," in *Proceedings of the 9th International Workshop on Software Specification and Design*, 1998, pp. 116–125.

[102] M. Zouari, C. Diop, and E. Exposito, "Multilevel and Coordinated Self-management in Autonomic Systems based on Service Bus," *Journal of Universal Computer Science (UCS)*, vol. 20, no. 3, pp. 431–460, 2014.

[103] J. Bourcier, "Auto-Home: une plate-forme pour la gestion autonomique dï¿½applications pervasives," Ph.D. dissertation, Université Joseph Fourier, 2008.

[104] T. Gu, H. Pung, and D. Zhang, "Toward an OSGi-based Infrastructure for Context-Aware Applications," *IEEE Pervasive Computing*, vol. 3, no. 4, pp. 66–74, 2004.

[105] D. Tkachenko, N. Kornet, E. Andrievsky, A. Lagunov, D. Kravtsov, and A. Kurbanow, "Management of IEEE 1394 Video Devices in OSGi Networks," in *Proceeding of the 10th IEEE International Symposium on Consumer Electronics (ISCE'06)*, 2006, pp. 1–6.

[106] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184–208, 2001.

[107] J. McAffer, P. VanderLei, and S. Archer, *OSGi and Equinox : Creating Highly Modular Java Systems*. Addison-Wesley, 2010.

[108] C.-S. D. Yang and L. L. Pollock, "Towards a structural load testing tool," in *ISSTA*, 1996, pp. 201–208.

[109] C. D. Grosso, G. Antoniol, M. D. Penta, P. Galinier, and E. Merlo, "Improving network applications security: a new heuristic to generate stress testing data," in *GECCO*. ACM, 2005, pp. 1037–1043.

[110] V. Garousi, L. C. Briand, and Y. Labiche, "Traffic-aware stress testing of distributed systems based on uml models," in *ICSE*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 391–400.

[111] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *ICSM*. IEEE, 2008, pp. 307–316.

[112] Z. M. Jiang, "Automated analysis of load testing results," in *Proceedings of ISSTA'10*. Trento, Italy: ACM, 12-16 July 2010, pp. 143–146.

[113] P. C. Jorgensen, *Software testing - a craftsman's approach (3. ed.).* Taylor & Francis, 2008.

[114] B. Beizer, *Software testing techniques (2. ed.).* Van Nostrand Reinhold, 1990.

[115] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[116] B. Beizer, *Software system testing and quality assurance.* New York, NY, USA: Van Nostrand Reinhold Co., 1984.

[117] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *ICSM.* IEEE, 2009, pp. 125–134.

[118] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *NSDI.* USENIX, 2006.

[119] J. Krizanic, A. Grguric, M. Mosmondor, and P. Lazarevski, "Load testing and performance monitoring tools in use with ajax based web applications," in *33rd International Convention on Information and Communication Technology, Electronics and Microelectronics.* Opatija, Croatia: IEEE, May 24 - 28 2010, pp. 428–434.

[120] A. Avritzer and B. Larson, "Load testing software using deterministic state testing," in *ISSTA*, 1993, pp. 82–88.

[121] A. Avritzer and E. J. Weyuker, "Generating test suites for software load testing," in *ISSTA*, 1994, pp. 44–57.

[122] ——, "The automatic generation of load test suites and the assessment of the resulting software," *IEEE Trans. Software Eng.*, vol. 21, no. 9, pp. 705–716, 1995.

[123] J. Zhang and S. C. Cheung, "Automated test case generation for the stress testing of multimedia systems," *Softw., Pract. Exper.*, vol. 32, no. 15, pp. 1411–1435, 2002.

[124] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *GECCO.* ACM, 2005, pp. 1021–1028.

[125] ——, "Using genetic algorithms for early schedulability analysis and stress testing in real-time systems," *Genetic Programming and Evolvable Machines*, vol. 7, no. 2, pp. 145–170, 2006.

[126] M. S. Bayan and J. W. Cangussu, "Automatic stress and load testing for embedded systems," in *COMPSAC (2).* IEEE Computer Society, 2006, pp. 229–233.

[127] X. Wang, B. Zhou, and W. Li, "Model based load testing of web applications," in *ISPA*. IEEE, 2010, pp. 483–490.

[128] A. J. Maâlej, M. Krichen, and M. Jmaïel, "Conformance testing of ws-bpel compositions under various load conditions," in *Proceedings of the 36th IEEE Annual International Computer Software and Applications Conference*. Izmir, Turkey: IEEE Computer Society, July 2012, p. 371.

[129] A. J. Maâlej, M. Hamza, and M. Krichen, "WSCLT: A tool for ws-bpel compositions load testing," in *Proceedings of the 22nd IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*. Hammamet, Tunisia: IEEE Computer Society, June 17-20 2013, pp. 272–277.

[130] A. J. Maâlej, M. Hamza, M. Krichen, and M. Jmaïel, "Automated significant load testing for ws-bpel compositions," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*. Luxembourg: IEEE Computer Society, March 18-22 2013, pp. 144–153.

[131] G. Canfora and M. Di Penta, "Testing services and service-centric systems: Challenges and opportunities," *IT Professional*, vol. 8, no. 2, pp. 10–17, Mar. 2006.

[132] C.-H. Liu, S.-L. Chen, and X.-Y. Li, "A ws-bpel based structural testing approach for web service compositions," in *Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering*, ser. SOSE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 135–141.

[133] G. Canfora and M. Penta, "Software engineering," A. Lucia and F. Ferrucci, Eds., 2009, ch. Service-Oriented Architectures Testing: A Survey, pp. 78–105.

[134] M. H. Mustafa Bozkurt and Y. Hassoun, "Testing web services: A survey," Department of Computer Science, King's College London, Tech. Rep. TR-10-01, January 2010.

[135] A. Bucchiarone, H. Melgratti, and F. Severoni, "Testing service composition," in *Proceedings of the 8th Argentine Symposium on Software Engineering*, Mar del Plata, Argentina, August 29-31 2007.

[136] Z. Zakaria, R. Atan, A. A. A. Ghani, and N. F. M. Sani, "Unit testing approaches for bpel: A systematic review," in *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*, ser. APSEC '09, 2009, pp. 316–322.

[137] H. M. Rusli, M. Puteh, S. Ibrahim, and S. G. H. Tabatabaei, "A comparative evaluation of state-of-the-art web service composition testing approaches," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: ACM, 2011, pp. 29–35.

[138] H. M. Rusli, S. Ibrahim, and M. Puteh, "Testing web services composition: A mapping study," *Communications of the IBIMA Journal*, vol. 2011, no. 598357, pp. 705–716, 2011.

[139] M. Petticrew and H. Roberts, *Systematic Reviews in the Social Sciences: A Practical Guide*. Blackwell Publishing, 2006.

[140] M. Mikucionis, K. G. Larsen, and B. Nielsen, "T-uppaal: Online model-based testing of real-time systems," in *Proceedings of ASE'04*. Linz, Austria: IEEE Computer Society, 20-25 September 2004, pp. 396–397.

[141] C. Barreto, V. Bullard, T. Erl, J. Evdemon, D. Jordan, K. Kand, D. Knig, S. Moser, R. Stout, R. Ten-Hove, I. Trickovic, D. van der Rijn, and A. Yiu, *Web Services Business Process Execution Language Version 2.0 Primer*, OASIS, May 2007.

[142] J. H. Hill, D. C. Schmidt, J. R. Edmondson, and A. S. Gokhale, "Tools for continuously evaluating distributed system qualities," *IEEE Software*, vol. 27, no. 4, pp. 65–71, 2010.

[143] O. Finkel, "Undecidable problems about timed automata," in *FORMATS'06*, ser. LNCS, vol. 4202, 2006, pp. 187–199.

[144] S. Tripakis, "Folk theorems on the determinization and minimization of timed automata," *Inf. Process. Lett.*, vol. 99, no. 6, pp. 222–226, 2006.

[145] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis, "Controller synthesis for timed automata," in *Proc. IFAC Symposium on System Structure and Control*. Elsevier, 1998.

[146] R. Alur, L. Fix, and T. Henzinger, "A determinizable class of timed automata," in *CAV'94*, ser. LNCS, vol. 818. Springer, 1994.

[147] P. V. Suman, P. K. Pandya, S. N. Krishna, and L. Manasa, "Timed automata with integer resets: Language inclusion and expressiveness," in *FORMATS*, ser. Lecture Notes in Computer Science, vol. 5215. Springer, 2008, pp. 78–92.

[148] P. Bouyer, F. Chevalier, and D. D'Souza, "Fault diagnosis using timed automata," in *FOSSACS'05*, ser. LNCS, vol. 3441, 2005, pp. 219–233.

[149] E. Grädel, W. Thomas, and T. Wilke, Eds., *Automata, Logics, and Infinite Games: A Guide to Current Research*. New York, NY, USA: Springer-Verlag New York, Inc., 2002.

[150] N. Bertrand, A. Stainer, T. Jéron, and M. Krichen, "A game approach to determinize timed automata," INRIA, Tech. Rep. 7381, september 2010, http://hal.inria.fr/inria-00524830.

[151] L. Manasa and S. N. Krishna, "Integer reset timed automata: Clock reduction and determinizability," *CoRR*, vol. abs/1001.1215, 2010.

[152] J. Schmaltz and J. Tretmans, "On conformance testing for timed systems," in *FORMATS'08*, ser. LNCS, vol. 5215, 2008, pp. 250–264.

[153] L. B. Briones and E. Brinksma, "A test generation framework for quiescent real-time systems," in *FATES'04*, ser. LNCS, vol. 3395, 2005, pp. 64–78.

[154] B. Nielsen and A. Skou, "Automated test generation from timed automata," *Software Tools for Technology Transfer*, vol. 5, no. 1, pp. 59–77, 2003.

[155] A. Khoumsi, T. Jéron, and H. Marchand, "Test cases generation for nondeterministic real-time systems," in *FATES'03*, ser. LNCS, vol. 2931, 2004, pp. 131–145.

[156] A. David, K. G. Larsen, S. Li, and B. Nielsen, "Timed testing under partial observability," in *ICST'09*. IEEE computer society, 2009, pp. 61 –70.

[157] N. Bertrand, A. Stainer, T. Jéron, and M. Krichen, "A game approach to determinize timed automata," in *FOSSACS'11*, 2011, to appear. Extended version as INRIA report 7381, http://hal.inria.fr/inria-00524830.

[158] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Timed I/O automata: a complete specification theory for real-time systems," in *HSCC'10*. ACM Press, 2010, pp. 91–100.

[159] C. Jard and T. Jéron, "TGV: theory, principles and algorithms," *Software Tools for Technology Transfer*, vol. 7, no. 4, pp. 297–315, 2005.

[160] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software - Concepts and Tools*, vol. 3, pp. 103–120, 1996.

[161] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi, "Alternating refinement relations," in *CONCUR'98*, ser. LNCS, vol. 1466, 1998, pp. 163–178.

[162] B. Bérard, P. Gastin, and A. Petit, "On the power of non-observable actions in timed automata," in *STACS'96*, ser. LNCS, vol. 1046, 1996, pp. 255–268.

[163] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using uppaal," in *Formal Methods and Testing*, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., 2008, pp. 77–117.

[164] M. Lahami, M. Krichen, and M. Jmaïel, "Safe and Efficient Runtime Testing Framework Applied in Dynamic and Distributed Systems," *Science of Computer Programming (SCP)*, vol. 122, no. C, pp. 1–28, 2016.

[165] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschnerr, "Model-based security testing: A taxonomy and systematic classification," *Softw. Test. Verif. Reliab.*, vol. 26, no. 2, pp. 119–148, Mar. 2016.

[166] A. Ahmad, F. Bouquet, E. Fourneret, F. L. Gall, and B. Legeard, "Model-based testing as a service for IoT platforms," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, 2016, pp. 727–742.

[167] O. Cheikhrouhou, "Secure group communication in wireless sensor networks: A survey," *J. Network and Computer Applications*, vol. 61, pp. 115–132, 2016.

[168] M. Krichen, "A formal framework for black-box conformance testing of distributed real-time systems," *IJCCBS*, vol. 3, no. 1/2, pp. 26–43, 2012.

[169] A. J. Maâlej and M. Krichen, "A model based approach to combine load and functional tests for service oriented architectures," in *Proceedings of the 10th Workshop on Verification and Evaluation of Computer and Communication System, VECoS 2016, Tunis, Tunisia, October 6-7, 2016.*, 2016, pp. 123–140.

[170] J. Gao, X. Bai, and W.-T. Tsai, "Cloud testing- issues, challenges, needs and practice," *Software Engineering : An International Journal (SEIJ)*, September 2011.

[171] X. Bai, M. Li, X. Huang, W. T. Tsai, and J. Gao, "Vee@cloud: The virtual test lab on the cloud," in *8th International Workshop on Automation of Software Test (AST)*, May 2013, pp. 15–18.

[172] K. Priyadarsini, V. Balasbramanian, and S. Karthik, "Cloud testing as a service," *International Journal Of Advanced Engineering Science And Technologies (IJAEST)*, 2011.

[173] L. Yu, L. Zhang, H. Xiang, Y. Su, W. Zhao, and J. Zhu, "A framework of testing as a service," in *2009 International Conference on Management and Service Science*, Sept 2009, pp. 1–4.

[174] L. Yu, W.-T. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, and W. Zhao, "Testing as a service over cloud," in *Proceedings of the Fifth IEEE International Symposium on Service Oriented System Engineering*, ser. SOSE '10, 2010, pp. 181–188.

[175] T. M. King and A. S. Ganti, "Migrating autonomic self-testing to the cloud," in *Third International Conference on Software Testing, Verification, and Validation Workshops*, April 2010, pp. 438–443.

[176] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *24th IEEE International Conference on Advanced Information Networking and Applications*, April 2010, pp. 27–33.

[177] P. Mell and T. Grance, "Draft nist working definition of cloud computing," 2009.

[178] L. M. Riungu, O. Taipale, and K. Smolander, "Software testing as an online service: Observations from practice," in *Third International Conference on Software Testing, Verification, and Validation Workshops*, April 2010, pp. 418–423.

[179] M. Lahami and M. Krichen, "Test Isolation Policy for Safe Runtime Validation of Evolvable Software Systems," in *Proceedings of the 22nd IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'13)*, 2013, pp. 377–382.

[180] IBM, "Infrastructure Optimization Services â" IBM Smart Business Test Cloud," http://www-935.ibm.com/services/us/en/it-services/systems/index.html.

[181] SOASTA, "Cloud Test by SOASTA," https://www.soasta.com/load-testing/.

[182] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, "Proactive Self-adaptation Under Uncertainty: A Probabilistic Model Checking Approach," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 1–12.

[183] H. W. Y. Adoni, T. Nahhal, M. Krichen, B. Aghezzaf, and A. Elbyed, "A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems," *Distributed and Parallel Databases*, pp. 1–36, 2019.

[184] M. Krichen, "Improving formal verification and testing techniques for internet of things and smart cities," *Mobile Networks and Applications*, pp. 1–12, 2019.

[185] ——, "Quelques Astuces pour Améliorer les Techniques de Vérification Formelle et de Test Basé sur des Modèles," Sep. 2019, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02289917

[186] M. Lahami, M. Krichen, and R. Alroobaea, "Tepaas: test execution platform as-a-service applied in the context of e-health," *International Journal of Autonomous and Adaptive Communications Systems*, vol. 12, no. 3, pp. 264–283, 2019.

[187] M. Krichen, A. J. Maâlej, and M. Lahami, "A model-based approach to combine conformance and load tests: an ehealth case study." *IJCCBS*, vol. 8, no. 3/4, pp. 282–310, 2018.

[188] M. Lahami, M. Krichen, and M. Jmaïel, "Runtime testing approach of structural adaptations for dynamic and distributed systems," *International Journal of Computer Applications in Technology*, vol. 51, no. 4, pp. 259–272, 2015.

[189] A. J. Maâlej and M. Krichen, "Study on the limitations of ws-bpel compositions under load conditions," *The Computer Journal*, vol. 58, no. 3, pp. 385–402, 2015.

[190] N. Bertrand, A. Stainer, T. Jéron, and M. Krichen, "A game approach to determinize timed automata," *Formal Methods in System Design*, vol. 46, no. 1, pp. 42–80, 2015.

[191] N. Bertrand, T. Jéron, A. Stainer, and M. Krichen, "Off-line test selection with test purposes for non-deterministic timed automata," *Logical Methods in Computer Science*, vol. 8, no. 4, 2012. [Online]. Available: https://doi.org/10.2168/LMCS-8(4:8)2012

[192] M. Krichen, "A formal framework for black-box conformance testing of distributed real-time systems," *International Journal of Critical Computer-Based Systems*, vol. 3, no. 1-2, pp. 26–43, 2012.

[193] Moez Krichen, "A black-box model-based framework for conformance testing of real-time distributed systems (in arabic)," 2018. [Online]. Available: http://rgdoi.net/10.13140/RG.2.2.22391.57764

[194] M. Lahami, M. Krichen, and M. Jmaiel, "A distributed test architecture for adaptable and distributed real-time systems," in *Avancées récentes dans le domaine des Architectures Logicielles : articles sélectionnés et étendus de CAL'2011, Lille, France, 7-8 Juin*

2011, ser. Revue des Nouvelles Technologies de l'Information, P. Aniorté, Ed., vol. L-6. Hermann, 2011, pp. 73–92. [Online]. Available: http://editions-rnti.fr/?inprocid=1001804

[195] S. Bensalem, M. Krichen, L. Majdoub, R. Robbana, and S. Tripakis, "A simplified approach for testing real-time systems based on action refinement," in *ISoLA 2007, Workshop On Leveraging Applications of Formal Methods, Verification and Validation, Poitiers-Futuroscope, France, December 12-14, 2007*, ser. Revue des Nouvelles Technologies de l'Information, Y. A. Ameur, F. Boniol, and V. Wiels, Eds., vol. RNTI-SM-1. Cépaduès-Éditions, 2007, pp. 191–202. [Online]. Available: http://editions-rnti.fr/?inprocid=1000545

[196] P. Bouyer, F. Chevalier, M. Krichen, and S. Tripakis, "Observation partielle des systèmes temporisés," *Journal européen des systèmes automatisés*, vol. 39, no. 1/3, p. 381, 2005.

[197] S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis, "Testing conformance of real-time applications by automatic generation of observers," *Electronic Notes in Theoretical Computer Science*, vol. 113, pp. 23–43, 2005.

[198] M. Krichen and M. Lahami, "Towards a runtime testing framework for dynamically adaptable internet of things networks in smart cities," in *Smart Infrastructure and Applications*. Springer, 2020, pp. 589–607.

[199] M. Krichen, M. Lahami, O. Cheikhrouhou, R. Alroobaea, and A. J. Maâlej, "Security testing of internet of things for smart city applications: A formal approach," in *Smart Infrastructure and Applications*. Springer, 2020, pp. 629–653.

[200] M. Krichen, "State identification," in *Model-based testing of reactive systems*. Springer, Berlin, Heidelberg, 2005, pp. 35–67.

[201] ——, "Testing real-time systems using determinization techniques for automata over timed domains," in *International Colloquium on Theoretical Aspects of Computing*. Springer, 2019, pp. 124–133.

[202] M. Krichen and R. Alroobaea, "A new model-based framework for testing security of iot systems in smart cities using attack trees and price timed automata," in *14th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE 2019*, 2019.

[203] M. Krichen, R. Alroobaea, and M. Lahami, "Towards a runtime standard-based testing

framework for dynamic distributed information systems," in *21st International Conference on Enterprise Information Systems - ICEIS 2019*, vol. 1, 2019.

[204] M. Lahami, M. Krichen, and R. Alroobaea, "Towards a test execution platform as-a-service: Application in the e-health domain," in *2018 International Conference on Control, Automation and Diagnosis (ICCAD)*. IEEE, 2018, pp. 1–6.

[205] M. Krichen, A. J. Maâlej, M. Lahami, and M. Jmaiel, "A resource-aware model-based framework for load testing of ws-bpel compositions," in *International Conference on Enterprise Information Systems*. Springer, Cham, 2018, pp. 130–157.

[206] A. J. Maâlej, M. Lahami, M. Krichen, and M. Jmaïel, "Distributed and resource-aware load testing of ws-bpel compositions." in *ICEIS (2)*, 2018, pp. 29–38.

[207] M. Krichen, O. Cheikhrouhou, M. Lahami, R. Alroobaea, and A. J. Maâlej, "Towards a model-based testing framework for the security of internet of things for smart city applications," in *International Conference on Smart Cities, Infrastructure, Technologies and Applications*. Springer, 2017, pp. 360–365.

[208] A. J. Maâlej, M. Krichen, and M. Jmaïel, "Wsclim: A tool for model-based testing of ws-bpel compositions under load conditions," in *International Conference on Tests and Proofs*. Springer, 2017, pp. 139–151.

[209] M. Lahami, M. Krichen, H. Barhoumi, and M. Jmaiel, "Selective test generation approach for testing dynamic behavioral adaptations," in *IFIP International Conference on Testing Software and Systems*. Springer, 2015, pp. 224–239.

[210] A. J. Maâlej, M. Hamza, and M. Krichen, "Wsclt: a tool for ws-bpel compositions load testing," in *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, 2013, pp. 272–277.

[211] N. Bertrand, A. Stainer, T. Jéron, and M. Krichen, "A game approach to determinize timed automata," in *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. Hofmann, Ed., vol. 6604. Springer, 2011, pp. 245–259. [Online]. Available: https://doi.org/10.1007/978-3-642-19805-2_17

[212] M. Krichen, "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)*, 2010.

[213] M. Lahami, M. Krichen, M. Jmaiel, and A. Idani, "A generic process to build reliable distributed software components from early to late stages of software development," in *The 2010 International Conference on Computer Engineering & Systems*. IEEE, 2010, pp. 287–292.

[214] M. Krichen and M. Solanki, "Automatic generation of realtime observers for monitoring web services," in *Proceedings of the Second International Conference on Web and Information Technologies (ICWIT'09)*, 2009.

[215] M. Gallien, F. Gargouri, I. Kahloul, M. Krichen, T.-H. Nguyen, S. Bensalem, and F. Ingrand, "Dâ$^{\mathrm{TM}}$une approche modulairea une approche orientée composant pour le développement de systemes autonomes: Défis et principes," *Proceedings of Control Architectures of Robots, CAR*, 2008.

[216] S. Bensalem, M. Bozga, M. Gallien, F. F. Ingrand, M. Krichen, and S. Tripakis, "Automatic generation of observers for the dala robot with ttg," in *AIP Conference Proceedings*, vol. 1019, no. 1. American Institute of Physics, 2008, pp. 487–492.

[217] M. Krichen and S. Tripakis, "Interesting properties of the conformance relation tioco," in *ICTAC'06*, 2006.

[218] ——, "An expressive and implementable formal framework for testing real-time systems," in *The 17th IFIP Intl. Conf. on Testing of Communicating Systems (TestCom'05)*, ser. LNCS, vol. 3502. Springer, 2005, available at http://www-verimag.imag.fr/PEOPLE/Stavros.Tripakis/papers/testcom05a.pdf.

[219] ——, "State identification problems for timed automata," in *The 17th IFIP Intl. Conf. on Testing of Communicating Systems (TestCom'05)*, ser. LNCS, vol. 3502. Springer, 2005, available at http://www-verimag.imag.fr/PEOPLE/Stavros.Tripakis/papers/testcom05b.pdf.

[220] ——, "Real-time testing with timed automata testers and coverage criteria," in *Formal Techniques, Modelling and Analysis of Timed and Fault Tolerant Systems (FORMATS-FTRTFT'04)*, ser. LNCS, vol. 3253. Springer, 2004, available as Verimag technical report TR-2004-15 at http://www-verimag.imag.fr/TR/TR-2004-15.pdf.

[221] A. J. Maâlej and M. Krichen, "A model based approach to combine load and functional tests for service oriented architectures." in *VECoS*, 2016, pp. 123–140.

[222] M. Lahami, M. Krichen, and M. Jmaïel, "Runtime Testing Framework for Improving Quality in Dynamic Service-based Systems," in *Proceedings of the 2nd International Workshop on Quality Assurance for Service-based Applications (QASBA'13), in conjunction with (ISSTA'13)*, 2013, pp. 17–24.

[223] A. J. Maâlej, Z. B. Makhlouf, M. Krichen, and M. Jmaiel, "Conformance testing for quality assurance of clustering architectures," in *Proceedings of the 2013 International Workshop on Quality Assurance for Service-based Applications*, 2013, pp. 9–16.

[224] A. J. Maâlej, M. Krichen, and M. Jmaïel, "Wscct: A tool for ws-bpel compositions conformance testing," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1055–1061.

[225] A. J. Maâlej, M. Krichen, and M. Jmaïel, "Model-Based Conformance Testing of WS-BPEL Compositions," in *Proceeding of the 4th IEEE International Workshop on Software Test Automation (STA'12) in conjunction with (COMPSAC '12)*, 2012, pp. 452–457.

[226] S. Bensalem, M. Krichen, and S. Tripakis, "State identification problems for input/output transition systems," in *2008 9th International Workshop on Discrete Event Systems*. IEEE, 2008, pp. 225–230.

[227] S. Bensalem, M. Krichen, L. Majdoub, R. Robbana, and S. Tripakis, "Test generation for duration systems," in *First International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2007) 1*, 2007, pp. 1–14.

[228] M. Krichen and S. Tripakis, "State identification problems for finite-state transducers," in *Formal Approaches to Testing and Runtime Verification (FATES-RV'06)*, ser. LNCS. Springer, 2006, to appear.

[229] S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis, "Testing conformance of real-time applications by automatic generation of observers," in *4th International Workshop on Runtime Verification (RV'04)*, ser. ENTCS, vol. 113. Elsevier, 2005, pp. 23–43.

[230] M. Krichen and S. Tripakis, "Black-box conformance testing for real-time systems," in *11th International SPIN Workshop on Model Checking of Software (SPIN'04)*, ser. LNCS, vol. 2989. Springer, 2004, available at http://www-verimag.imag.fr/PEOPLE/Stavros.Tripakis/papers/timetest.pdf.