Test case generation for Symbolic Distributed System Models : Application to Trickle based IoT Protocol

Ngo Minh Thang Nguyen

université PARIS-SACLAY

CentraleSupélec

NNT : 2019SACLC092

Thèse de doctorat

Test case generation for Symbolic Distributed System Models: Application to Trickle based IoT Protocol

Thèse de doctorat de l'Université Paris-Saclay
préparée à CentraleSupélec

École doctorale n°573 Approches Interdisciplinaires: Fondements, Applications et Innovation (Interfaces)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Gif-sur-Yvette, le 16/12/2019, par

NGO MINH THANG NGUYEN

Composition du Jury :

Stéphane MAAG Professeur, Telecom SudParis	Président
Antoine ROLLET Maître de conférences, Université de Bordeaux	Rapporteur
Frédéric MALLET Professeur des universités, Université Côte d'Azur	Rapporteur
Pascale LE GALL Professeur des universités, CentraleSupélec	Directeur de thèse
Boutheïna BANNOUR Ingénieur chercheur, CEA LIST	Co-encadrante de thèse

Acknowledgements

This work has been carried out within the Laboratory of System Requirements and Conformity (LECS) and the Laboratory of Mathematics and Computing for Complexity and Systems (MICS) at respectively the Alternative Energies and Atomic Energy Commission and CentraleSupelec. It is funded by the Laboratory LECS.

This work has been accomplished under the supervision of Professor Pascale Le Gall (MICS) and my supervisor Boutheïna Bannour (LECS). I would like to express my deepest acknowledgment to them and their highly professional guidance, dedication and incessant support. I highly appreciate the friendly and professional environment they have created for me during my doctorate. They have always been available whenever I had a question, and always interested in the progress of my work. The discussions we have had, and their advice, have always allowed me to progress. Finally, their proofreading to this thesis were very precious.

I would like to show especially my gratitude to Arnault Lapitre (LECS) and Paolo Ballarini (MICS) for their help and valuable advice during my doctorate. Their rigors, their problem solving skills and their knowledge have always been helpful to me.

I sincerely thank the jury: Mr. Antoine Rollet and Mr. Frédéric Mallet for accepting to review my PhD thesis. I also would like to extend my special thanks to Mr. Stéphane Maag for agreeing to examine this thesis.

My thanks go also to my doctoral friends in CEA: Lamia, Hadi, Slim and Imen in my laboratory and Lionel, Jean-Christophe in the Laboratory LSL. I would like to thank Frédérique Descreaux, the secretary of my laboratory for her assistance. I wish to thank my colleagues with whom I had lunch, shared jokes and played football together: Jean-Yves, Xavier, Alain, Julien, Virgile and Adel.

Finally, I owe unconditional thanks for the support of my family. I will never forget helpful advice of my aunt Lien, my aunt Hang and my uncle Thong. Above all, I would like to thank my mom My, my dad Phap, my sister Dung, my nephew Subeo and my beloved Thao Nguyen for their unfailing support and continuous encouragement throughout my doctoral years.

Résumé étendu en français

Les systèmes distribués sont composés de nombreux sous-systèmes distants les uns des autres. Afin de réaliser une tâche commune, les sous-systèmes communiquent à la fois avec l'environnement local par des messages qualifiés d'externes et avec d'autres sous-systèmes par des messages qualifiés d'internes grâce à un réseau de communication. En pratique, les systèmes distribués mettent en jeu plusieurs types d'erreurs, soit provenant des sous-systèmes les composant, soit provenant des messages internes. Cela explique l'importance de tester de tels systèmes afin d'augmenter le niveau de confiance en leur bon fonctionnement. Cependant, en raison même de leurs caractéristiques, il est reconnu qu'il est compliqué de tester les systèmes distribués. Sans horloge globale, les sous-systèmes ne peuvent pas synchroniser les messages, ce qui entraîne des situations non déterministes, difficiles à contrôler.

L'approche de test, appelée test à base de modèles (MBT pour Model Based Testing) a pour objectif de vérifier si le comportement d'un système sous test (SUT) est conforme à son modèle, en charge de décrire les comportements attendus du système. Les approches de test à base de modèles ont un rôle majeur pour 2 étapes principales du test : la génération de cas de test et le calcul de verdict. Dans cette thèse, nous nous intéressons à la génération de cas de test locaux dans le contexte du test des systèmes distribués : les cas de test sont coordonnés dans la mesure où chacun d'entre eux est paramétré par un objectif de test local obtenu par projection d'un objectif de test global associé au système global.

Pour spécifier les systèmes, nous utilisons les systèmes de transition symbolique temporisé à entrées et sorties (TIOSTS), qui sont des automates communicants équipés de techniques d'exécution symbolique portant sur les données et le temps. Le modèle d'un système distribué est un tuple de TIOSTS, chacun décrivant un sous-système et la communication entre sous-systèmes étant modélisée par les files d'attente. Nous utilisons la technique d'exécution symbolique pour décrire les comportements symboliques du système distribué. Faisant partie des hypothèses de test adoptées dans ce manuscrit, on suppose qu'en plus des messages externes, un cas de test local observe les messages internes reçus et envoyés par le sous-système co-localisé. Cette hypothèse réduit le problème de contrôlabilité puisqu'en observant ces messages, un cas de test local peut connaître l'ordre d'émission et de réception des messages internes du sous-système co-localisé. Notre framework de test est constitué des étapes suivantes : dans un premier temps, on sélectionne un objectif de test global, qui correspond à un comportement particulier dans le système distribué, obtenu via des techniques d'exécution symbolique. Deuxièmement, les techniques de projection sont appliquées pour obtenir des objectifs de test locaux à partir de l'objectif de test global précédemment sélectionné. Troisièmement, nous décrivons les cas de test locaux à l'aide de règles indiquant dans quelles circonstances le cas de test stimule le système local sous test avec une donnée de test ou bien attend d'observer une réaction du sous-système, tout en suivant l'objectif de test local correspondant. L'exécution distribuée du test consiste à exécuter des cas de test locaux sur chacun des sous-systèmes distants : les données de test sont calculées à l'aide de techniques de résolution de contraintes et les verdicts de test sont calculés à la volée.

Une implémentation de notre framework de test est proposée au sein de Diversity, qui est un outil d'analyses formelles basées sur un modèle utilisant l'exécution symbolique. Cet outil personnalisable nous facilite l'implémentation de nouvelles fonctionnalités grâce à une architecture extensible du moteur de l'exécution

symbolique. L'utilisation de notre framework de test dans Diversity se compose de deux étapes: sélection d'un objectif de test global et dérivation de cas de test locaux à partir de l'objectif de test sélectionné. Dans le but d'expérimenter notre framework de test, nous le mettons en œuvre sur un cas d'étude issu d'un protocole de communication utilisé dans le contexte des systèmes IoT. Enfin, une méthode de sélection est proposée pour sélectionner des objectifs de test intéressants à tester avec ce cas d'étude.

Mots clés: Systèmes distribués, test à base de modèles, systèmes de transition symbolique temporisé avec des entrées et sorties, exécution symbolique, objectif de test global, technique de projection, objectif de test local, résolution de contraintes.

Contents

Chapter 1

Introduction

Contents

1.1 Distributed systems and motivation

Distributed systems (DS) [21, 76] can be considered as a collection of subsystems that in order to achieve a common goal coordinate their actions by exchanging messages through an interconnected communication network. Each subsystem is autonomous and interacts with its local environment through external channels and with other subsystems through internal channels. Several concrete examples of DS can be found in different domains, including: communication networks, e.g. Local Area Networks (LAN) which connect thousands of servers in data centers; Internet of Things (IoT) networks which exchange application data and are applied to domains such as agriculture, health and smart home; real time process controls, e.g. aircraft control systems which are used to manage flight routes for takeoff and landing.

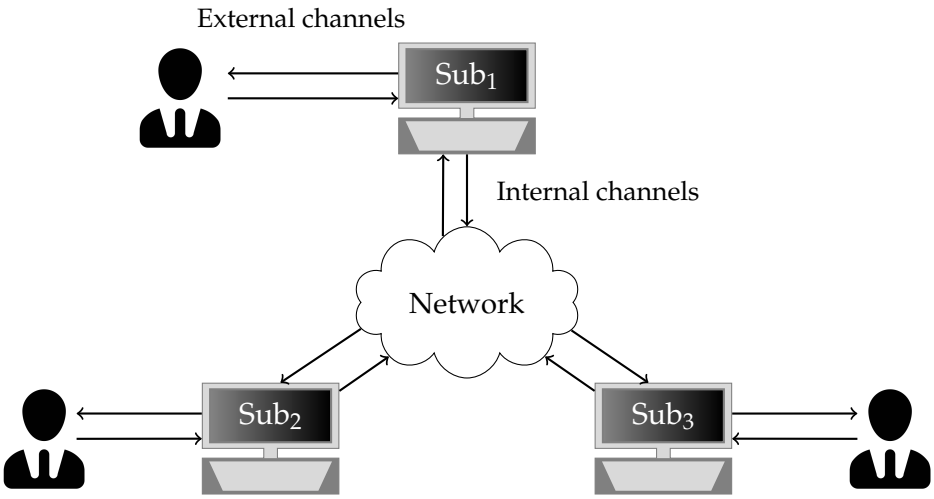


FIGURE 1.1: Overview of a distributed system

Figure 1.1 schematically depicts a distributed system, generically called *Sys* with three subsystems Sub_1, Sub_2 and Sub_3. Each of the subsystems Sub_i can communicate with the other subsystems Sub_j (for $j \neq i$) via internal channels (black lines going to the cloud pattern) as well as with its direct environment via external channels (black lines going to the black silhouette, schematizing any user, either an human being or a machine.

In spite of their widespread diffusion, the conception and deployment of DS is affected by a number of critical aspects [55], the most relevant of which is the lack of a global clock which may induce non-deterministic interleaving of exchanged messages, hence letting the overall behavior of DS non-deterministic. Indeed, without some mechanisms playing the role of a global clock, distributed systems are unable to guarantee that handling of received messages is correct w.r.t. their emission date. For instance, it suffices that the order of two receptions in a subsystem has been reversed to yield a flow of messages which would not occur if the messages were received in the correct order.

DS often entail typical errors which roughly speaking, may be split in two categories: errors due to non-compliance with some real-time constraints as opposed to errors affecting data included in the messages exchanged by the subsystems. Errors of the first category arise when the correct operation of the system requires, for example, that the reply to a request received by Sub_1 must be issued within a given delay. Exceeding such delay should be considered as a display of a misbehavior which could be due to either: i) an erroneous implementation of Sub_1; ii) a misbehaving of the other subsystems (Sub_2 and Sub_3); iii) an excessive communication's latency due to insufficient Quality of Service (QoS) of the underling network. On the other hand, errors of the second category entail situations where an unexpected data is carried over a channel, and hence as to be considered as invalid. Such kind of errors may result from faulty internal communications, e.g. an internal message could be dropped or be corrupted, leading to a situation where the message has been emitted by a subsystem but never received by the corresponding receiver. Obviously, an erroneous exchanged data can simply come from the subsystem that emits the data, and therefore can come directly from an erroneous subsystem. To summarise, errors in the context of distributed systems may be caused either by errors of their subsystems, by errors of the underlying communication network, or by a combination of both. In this respect, testing of distributed systems requires taking into account sufficiently sophisticated methods capable of dealing with all these kinds of errors.

Many kinds of testing methodologies have been proposed in the literature [60], arising at different levels in the development cycle of a computerized system. Unit testing [24, 17] is a process concerned with testing of individual parts of a system, considered *in isolation*, i.e. without accounting for any interactions with other components. In the context of distributed systems, whereby a subsystem may be considered as an individual part (hence its local behavior could be tested through unit testing), integration testing [67] represents the second level of testing, which aims at detecting errors due to the interactions between the subsystems that form a DS. Therefore DS testing consists of verifying the compliance with some (global) specifications of the entire system (i.e. the system consisting of all interacting subsystems). In some cases, the knowledge of the DS architecture and/or of the structure of the (global) specification can facilitate testing activities, in particular through the adoption of a compositional approach.

In this context, the objective of this thesis is to propose an integrated testing framework that allows one for verifying whether a DS composed of a number of subsystems operates correctly w.r.t its specification, where a DS specification consists of a set of specifications, one for each subsystem composing it. In the next section, we overview several testing architectures that have been introduced in the literature for testing distributed systems.

1.2 Testing architectures and related works

General architecture. Figure 1.2 illustrates a general testing architecture for distributed systems. It accounts for different possible configurations that can be considered during a testing process, such as:

1. At every subsystem, we place a local tester (depicted as a dummy holding a magnifying glass) in order to test it locally;
2. Local testers could coordinate each other (represented by dotted green arrows) in order to synchronize their behaviors for testing a global behavior (i.e. a behavior that accounts for the combination of several subsystems behaviors);
3. A local tester may observe internal communications (represented by dotted blue arrow) which are sent/received by its corresponding subsystem
4. A global tester (dummy with a magnifying glass linked through dotted pink arrows to local tester's dummies) connected to all local testers is in charge of i) collecting local traces together with their associated local verdict or ii) controlling the process such as beginning or terminating the whole process or iii) analysing the compatibility of all local traces up to internal communication rules or iv) emitting a global verdict from the knowledge of all local verdicts together with the analysis of internal communications etc.

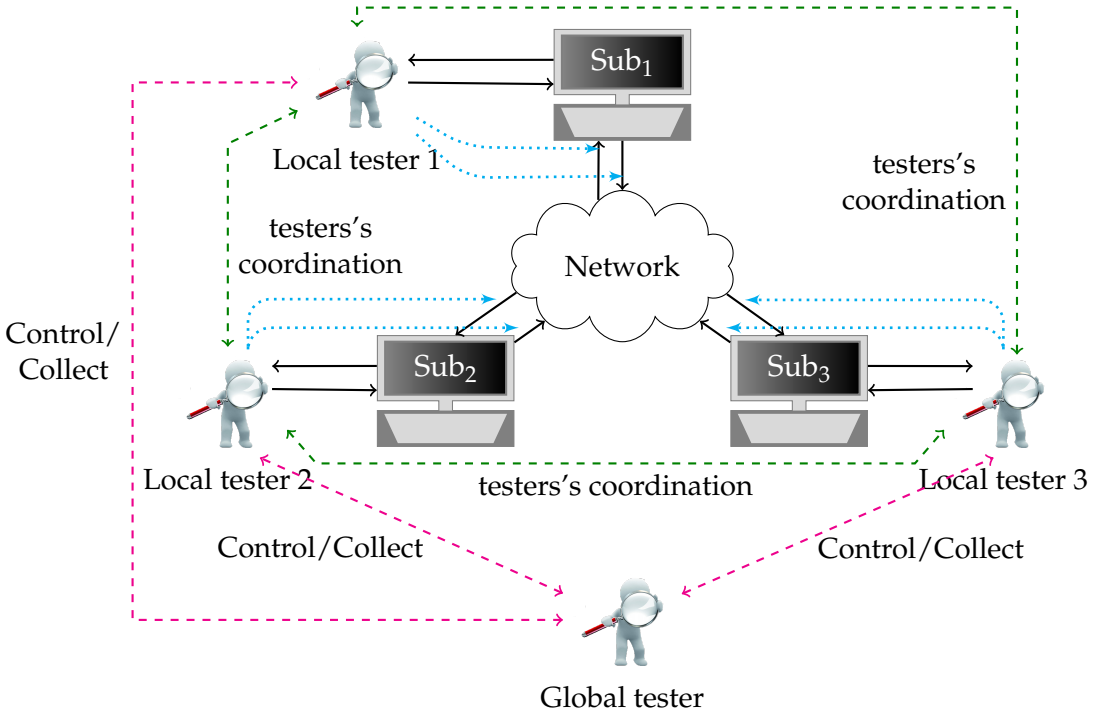


FIGURE 1.2: General architecture

According to state-of-the-art literature [82, 45] a testing architecture for distributed systems may be referred to as: global-tester based architecture, local-tester based architecture and hybrid architecture. Let us briefly introduce these kinds of testing architectures.

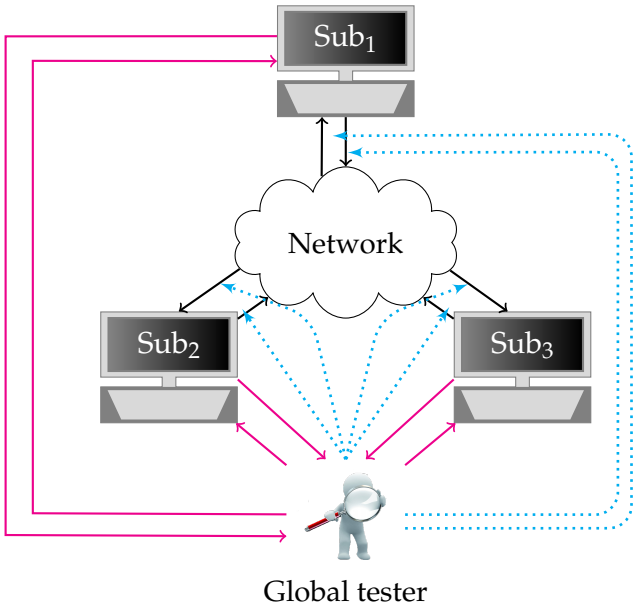


FIGURE 1.3: Global-tester based architecture

Global-tester based architecture. A global tester (Figure 1.3) is characterized by two main features: he/she is capable of directly interacting with every subsystem and also has access to the internal communications of each subsystem. As a consequence, in order to compute a test verdict, the global tester can: i) stimulate and collect the resulting messages at each subsystem (depicted by magenta arrow); ii) observe the content of internal channels of each components of the DS (depicted by cyan dashed arrow).

The most important advantage of this architecture is that the global tester has an overview of global behaviors and controls easily the stimulus to drive a global behavior. However, in this architecture, it is required that the tester possesses a strict control over the system execution so as to control the global behavior that is being tested. For instance, an unstable connection between the global tester and a subsystem can delay a stimulus, hence resulting in reversing the order of messages (i.e. test actions) in the system execution. Furthermore the inherent latency which affects the retrieval of data collected by the global tester from the subsystems being tested makes it difficult to accurately reconstruct the sequence of events in the system, and therefore to know whether the overall reconstructed behavior is in line with expectations. The consideration of test data delays between remote points of observations and a centralized global tester is known as the main difficulty of remote testing, often partially resolved by assuming that these transmission delays are bounded by a known boundary.This issue is not present in the following testing architecture where each subsystem is equipped with a local tester, under the testing hypothesis that local testers synchronously communicate with the co-localized subsystems.

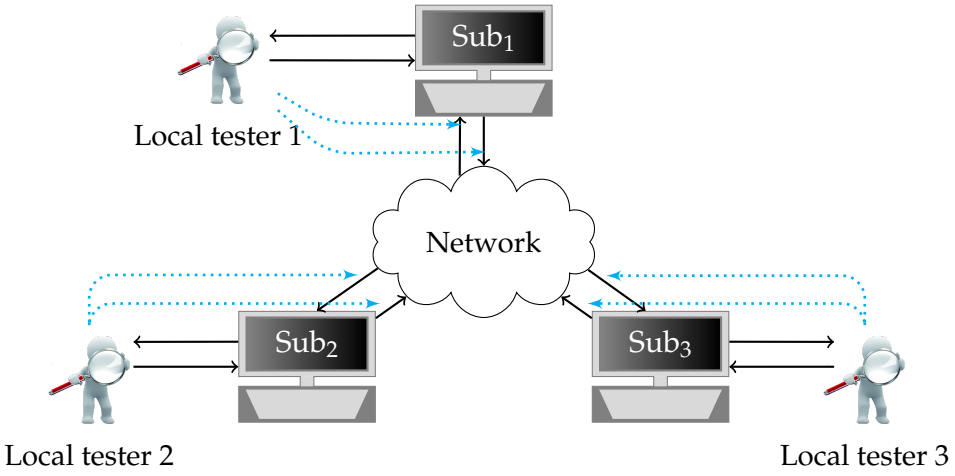


FIGURE 1.4: Local-tester based architecture

Local-tester based architecture. A local-tester architecture consists in decomposing the global testing goal (i.e. the desired test of the entire DS) into a number of local goals, each of which is to be performed by a dedicated local tester assigned to a specific subsystem (see Figure 1.4). Testing of the DS in this case consists of the composition of the activity that each local tester performs independently on the corresponding subsystem. Local testers may observe internal messages sent/received to/from their corresponding subsystems. Furthermore stimulus applied by a local tester undergoes no delay to be reached by the corresponding subsystem (since each local tester interacts directly with its subsystem). A shortcoming of the local-tester architecture is that since one does not have a global view, it is not possible for local testers to control sending of internal messages to meet a targeted global behavior.

Hybrid architecture. The hybrid testing architecture consists of a combination of the global-tester and local-tester architecture. With the hybrid approach a tester may be associated with one or several subsystems as depicted in Figure 1.5 whereby local testers 1 and 4 are associated with subsystem Sub_1 while subsystems Sub_1 and Sub_2 are connected to the local tester 1.

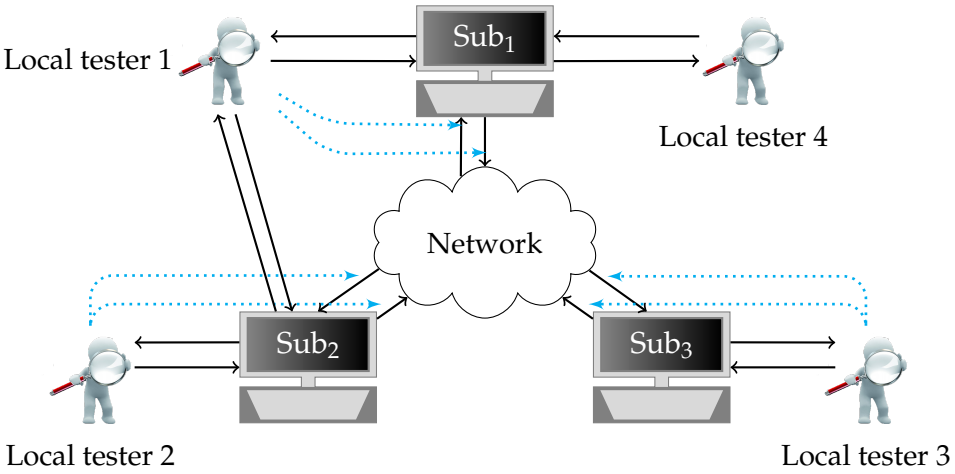


FIGURE 1.5: Hybrid architecture

Model-Based Testing. The literature is populated with a large number of methods devoted to testing of distributed systems. Approaches based on the application of some formal modelling paradigm (such as, e.g., state transition machines, data flow or control flow models, etc.) to express a system (correct) behavior, commonly known as system *specification*, belong to the family of so-called Model-Based Testing (MBT) methods. Roughly speaking, MBT characterises a family of testing approaches where models are used to derive test cases whose execution is used to compute verdicts. Specifically: the runtime behavior of a system is tested through tests that are devised from a specification expressed in terms of a (formal) model. MBT is very often studied in order to efficiently test a system [72, 43, 35, 65, 64, 62]: From a predefined model which characterizes expected behaviors, one can derive test cases according to a test purpose describing a particular behavior. The generated test cases are then executed on the actual system (i.e. test execution) and results are collected and analysed so to produce a *test verdict*. The main objective of MBT is to test that the system behaviors do not violate behaviors expressed by the model. Thus, the last step consists in analyzing the result of test execution so to conclude if the system behaves as expected. If the outcome of a test execution diverges from those specified in the model, an error is identified and the system behavior is suspect. The testing approach for DS presented in this dissertation is based on MBT.

Conformance relations. The process of testing a system is concerned with establishing whether a system behaves correctly (w.r.t. some specifications). In the context of system testing the notion of system's correctness is usually given in mathematical terms through a relation defined w.r.t. the traces (i.e. sequences of input and output) observed on the *system under test* (SUT).
In the class of MBT based on Finite State Machines (FSM) models, i.e. models for which a (possibly empty) output is systematically associated to an input, conformance is often expressed in term of distinguishing sequences and of coverage criteria for the underlying model, so that a successful test campaign consists in the generation of test sequences up to some coverage criteria on states or transitions.
When considering models in the class of labelled transition systems, for which outputs are not systematically coupled with inputs, establishing the correctness of the SUT is based on the so-called notion of *conformance relation*. The *input output conformance relation* (ioco) [78, 77] is the original conformance relation upon which much of MBT literature of the 90s is based. It has then been extended by many other variants to adapt it to different kind of models, such as timed variant tioco [46, 73, 45, 5] for timed systems, dtioco for distributed timed systems [32, 10]. In the rest of the document, we will place ourselves within the framework of models in the form of labelled transition systems provided with conformance relations issued from the *ioco* relation family.

Background of our work. The work presented in this document is a complementary contribution to that described in [32], [10]. Indeed, in [10], authors consider a particular testing architecture for which a local tester is associated to each subsystem and for which a global tester collects all local behaviors for *a posteriori* analysis (Figure 1.6). In particular, in addition to external communications, local testers are supposed to observe local internal communications. By interacting with the corresponding subsystem, the local tester collects a local trace and sends it to the global tester. More precisely, as the correctness of a distributed system is defined in a modular way, i.e. by the (local) correctness of each local subsystem with respect to its local specification, and by the correctness of internal communications with respect to communication rules. These communication rules governing internal communications simply express that a message, before it is received, has first been sent, that a message may take an arbitrarily long time to reach its destination. In [10], the authors have modeled these rules as a constraint solving problem. Moreover, since each of the local traces is checked in an off-line mode against its local specification, the approach of [10] has the advantage of proposing a test method structured according to the structure of the system itself. On the other hand it has the disadvantage of not providing the tester with any means to control the behavior that are analysed. Indeed, the coordination

of local testers is not taken into consideration in this approach. In this document, we propose to remedy this shortcoming by assuming the same system structure and testing architectures as in [10], and yet seeking a method that allows for controlling the behavior executed during the testing phase, in particular by selecting test data from some global behaviors deemed relevant by the tester. In the following we summarise the main elements of the DS test framework defined in [10].

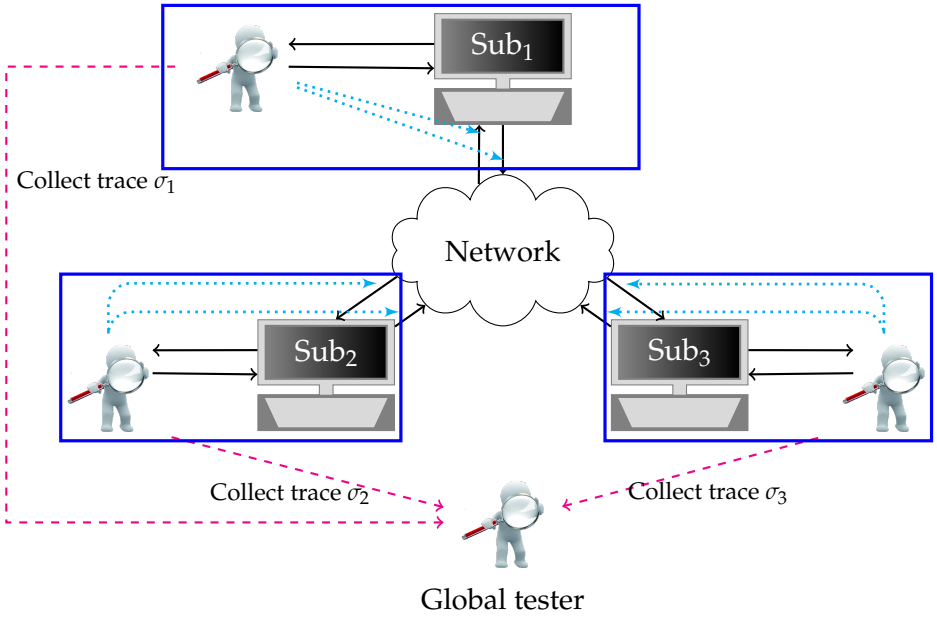


FIGURE 1.6: Testing architecture in the work [10]

In the approach of [10], the specification of the distributed system is defined as a tuple of three models where each of them specifies behaviors of a subsystem, i.e. $Sys = (M_1, M_2, M_3)$. [10] highlights a compositional result: the conformance of a DS is defined as the composition of local conformance for all subsystems and correctness of internal communications. In order to be able to derive the global test verdict, the tuple of local traces, $(\sigma_1, \sigma_2, \sigma_3)$ one for each subsystem, is verified as follows:

- Local conformance: each local trace σ_i for $1 \leq i \leq 3$ is checked w.r.t its corresponding model M_i in the context of the conformance relation tioco;
- Verification of internal communications: the tuple $(\sigma_1, \sigma_2, \sigma_3)$ of the three local traces is then analysed to check whether they are compatible to the broadcast communication policy : the emission date of an internal message must occur before the corresponding reception date for any other subsystem likely to receive the message. Each of these inequalities constitutes a constraint, element of the set of constraints that needs to be satisfied by the tuple of local traces. To do so, all the dates are fictitiously measured against a common initial date: as it is not possible to precisely synchronize the beginning of all traces together, a variable x_i is introduced per subsystem Sub_i, representing the time elapsed since an initial date common to all the subsystems, but unknown. Moreover, time is assumed to elapse at the same speed for all subsystems, so that time unit is the same for all subsystems. Moreover, no constraints are expressed on transmission delays in the internal network so that a message reception can take place after an arbitrarily long delay w.r.t. its corresponding emission. Under these hypotheses, the analysis of the correctness of internal communications can be expressed as a Constraint Satisfaction Problem [7, 81] with exactly n variables (the initial unknown duration x_i for each subsystem), whose solution can be obtained through a standard constraint solver (CVC4 [6], Z3 [25], Yices [28]).

In the best case, the local conformance in [10] is inherited from the conformance verification proposed by Bannour *et al.* in [5]: a sequence of inputs interleaved with delays is derived from a target behavior to be tested, and then is executed on the local subsystem. As we are in a high level non-deterministic context, the execution of a predefined selected timed input sequence is very likely to deviate from the initial selected behavior. Thus, while the approach described in [10] allows the tester to check whether a set of local traces is consistent with the conformance relation *dtioco*, (i.e. the *ioco* variant adapted to such timed distributed, combining *tioco* for each subsystem and internal communication rules), [10] does not propose a methodology to manage the submission of test data. Unlike [5], we aim at proposing a test method that deploys local testers who are able to submit input data based on both test purposes to be followed and previous observations on local subsystems. Such test approaches where local testers can control local testing process by choosing at which moment test data should be submitted are said to be *online* in contrast to *offline* approaches where input test data sequences are pre-computed.

In the following section, we present the initial assumptions concerning the testing architecture and our approach on which our work will be based.

1.3 Our proposed architecture and approach

Models of distributed systems. In our approach, to specify behaviors of a DS, we use a collection of models called Timed Input Output Symbolic Transition Systems (denoted as TIOSTS) [5, 48]. These are communicating automata which interact with the environment by receiving and sending messages through communication channels. In addition, transitions of these automata are symbolic ones and data exchanges on the channels and time are symbolic parameters. Guards and substitutions are constructed on these symbolic parameters. Each subsystem of a DS is modeled by a TIOSTS, communicating with other subsystems through waiting queues.

Symbolic execution techniques. To obtain behaviors of a distributed system, symbolic execution is considered as an interesting technique [44]. Indeed, from its model, we can execute it to obtain all possible symbolic behaviors. Furthermore, the particularity of symbolic execution is to explore paths of the model by using symbolic parameters (not concrete numerical values). During this exploration, the logical constraints are also computed to assure the feasibility of exploration paths. These constraints are constructed over symbolic parameters at each execution step. The result of symbolic execution can be represented as the form of a symbolic tree whose nodes are called *execution contexts* containing pieces of information related to the symbolic execution.

Test purpose. A test purpose is used to characterize a particular behavior to be relevant for testing. Concretely, a test purpose in our approach is issued from a symbolic behavior in the symbolic execution of system.

In our architecture (figure 1.7), each local test case is placed in direct interaction with its local subsystem. We do not consider coordination messages between test cases, once the testing process has been initiated. However, we have a global tester to initialize and terminate the whole testing process. Similarly to the work [10], we allow that a test case can observe local internal communications. From a global test purpose that will be selected among symbolic paths of the overall system, one can derive local test purposes that local test cases will have to follow. From each local test purpose, a test case is generated for the corresponding subsystem. This test case is an executable TIOSTS and is directly executed on the subsystem: the local test case can observe the external and internal communications from its corresponding subsystem and stimulate it with inputs. Unlike the previous architecture while local testers collect traces for a posterior verification, in

our approach, the local test case benefits from constraint solving techniques to compute next inputs based on previous observations, allowing to pilot the local test purpose.

Experimentation. For experimental purposes, our reference models are implemented in our platform called Diversity, which is a multipurpose machinery used to debug models and perform different analyses such as trace generation, deadlock detection or behavior selection. Moreover, being customizable, Diversity allows advanced users to implement their own analysis thanks to a dedicated process.

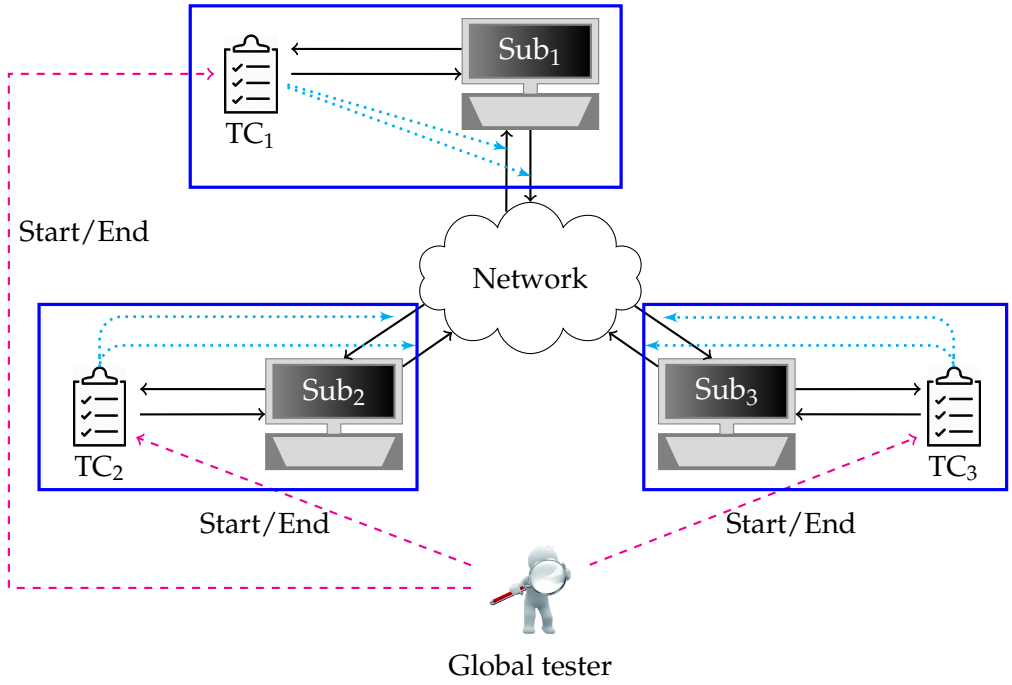


FIGURE 1.7: Our testing architecture

1.4 Contributions

We present in this dissertation the following contributions:

- We propose an online testing framework for subsystems in the distributed context. The derivation of a test case for a subsystem consists in constructing an automaton which can either observe messages emitted by the subsystem to compute a test verdict, or stimulate the subsystem with messages satisfying the test purpose.
- The testing framework for a distributed system is adapted from the testing framework for subsystems. Indeed, from a global test purpose, one can derive a local test purpose that defines local behaviors for each subsystem. Then, the testing framework consists in generating for each subsystem a local test case based on the corresponding local test purpose. Because of the distributed architecture, each local test case interacts only with the co-localized subsystem and is unconcerned with the execution of distant local test cases. At the end of the process, a global verdict is computed by aggregation of local test verdicts and the communication correctness of local traces [10];
- Our last contribution consists in providing selection criteria to compute global test purposes which are relevant for testing in a representative case study, which is a protocol for information dissemination in

a Wireless Sensor Network (abbreviated as WSN). This network applies Multicast Protocol for Low-Power and Lossy Networks (MPL for short) to quickly spread information for the purpose of keeping updated all sensors. It might also be noted that we benefit from the symbolic exploration in Diversity to implement the Send Receive Pair Coverage criterion (SRPC for short) [68, 69] as a dynamic selection heuristics;

As a part of our contributions, our testing framework is implemented in the Diversity tool to validate the proposed theory. This implementation of our testing framework is composed of the two following steps: selection of a global test purpose (characterized by a sequence of consecutive transitions) by checking its feasibility and derivation of local test cases. Furthermore, in order to facilitate the selection of a test purpose, we propose a novel means to visualize a scenario in distributed systems obtained from selection criteria. The new visualization has the form of a sequence diagram annotated with time and data information and can be readable by the PlantUML tool (`http://plantuml.com`).

1.5 Dissertation structure

In conformity with our contributions that have been presented above, we propose the following organisation of our dissertation:

- In Chapter 2, we present the symbolic model TIOSTS. Then, we introduce deterministic TIOSTS, a subclass of TIOSTS which is considered in our testing framework. We present also notations that will be used throughout the manuscript, and introduce small illustrative examples. This chapter is also devoted to the presentation of the Diversity tool as well as the underlying language used in this tool and the implementation of the test purpose coverage;
- Chapter 3 discusses firstly two principal testing approaches in MBT: offline and online testing. Then, we propose our online testing approach for a subsystem in the distributed context. This chapter provides also the set of construction rules to derive a test case for the subsystem. The next part is dedicated to the implementation of the test case generation in Diversity and elements for validation of our testing framework based on the conformance relation tioco. We also evaluate our implementation with respect to the test purposes of long size by doing experiments. Finally, we review the state of the art relevant to the context of unitary testing;
- In Chapter 4, we introduce distributed systems and the interest of testing them. Then, our testing architecture for distributed systems is discussed. In order to obtain behaviors of distributed systems, we define symbolic execution technique for DS which adds means to handle internal messages between subsystems and especially to reason about their causality using durations. The subsequent section is devoted to our testing framework for DS. Finally, we present our scenario generator to provide a new visualization of a test purpose for DS in the form of a sequence diagram;
- Chapter 5 presents selection criteria applied to the case study Trickle-based communication protocol MPL. From the specification of the protocol, we will construct timed symbolic models. Then, from the constructed models, we propose selection method to cover global test purposes which are relevant for testing in a WSN applying MPL. The coverage is based on the proposed selection method by taking advantage of the heuristics Hit-or-Jump [15] implemented in the symbolic execution of Diversity;
- Finally, Chapter 6 concludes the dissertation, as well as reviews the contributions of our approach and identifies perspectives that require further exploration in the future.

Chapter 2

Symbolic models for reactive systems

Contents

In this Chapter, we present Timed Input Output Symbolic Transition Systems (TIOSTS) [4, 5] for the purpose of modeling behaviors of reactive systems together with their dedicated symbolic execution techniques. These are perquisites to the symbolic framework that we develop in this thesis. This chapter is structured as follows:

- Section 2.1 describes the context of modeling reactive systems and introduces briefly several existing symbolic frameworks available in the literature;
- In Section 2.2, we present the formalism which is used throughout this thesis through two different issues: theory and implementation. The running example called ATM-Bank system that is used in this thesis is also introduced;
- Then, symbolic execution techniques are explained in Section 2.3. After talking about their principles, we introduce the enrichment by quiescence, a technique permitting to specify the situations when a system cannot produce an output;

- Section 2.4 is dedicated to the implementation of the model of ATM-Bank system in the Diversity tool. We introduce also an analysis module called TESTPURPOSE_SEL, a coverage strategy to verify the feasibility of a sequence of consecutive transitions. This module has been implemented in Diversity to support our approach;
- Finally, we conclude this chapter in Section 2.5.

2.1 Reactive systems and real-time systems

2.1.1 Reactive systems and real-time systems.

A reactive system is a system that interacts continuously with its environment through a well-identified interface as illustrated in Figure 2.1. Such an interface defines which kind of possible inputs (or stimulations) the system accepts and which possible outputs (or reactions) it can produce. The behavior of this kind of systems is not reduced to a simple transformation function applied to an input to generate an output. A behavior is an interleaving of inputs and outputs, with different durations between two consecutive actions, that can be indifferently inputs or outputs. That is, in addition to the last inputs that the system has received, the computation of the next output may depend on the state of the system being defined by the history of prior interactions so-far. Consequently, the behavior of a reactive system may be characterized by sequences of inputs and outputs separated by durations. Those sequences allow to reason about correctness of reactive systems: intuitively, a sequence reflects a legal execution of a reactive system provided that in case it receives legal inputs, the reactive system provides appropriate outputs as expected. In addition, some reactive systems are subject to timing constraints. They have to generate outputs in response to inputs, also called stimulations, within an acceptable duration. In other words, the absence or a late (arrival out of time) response is considered as serious as an erroneous output. They are part of the so-called real-time systems. Usually the moment of an event (input or output) occurrence is measured (timestamps, logs with dates...) allowing the definition of legal durations separating subsequent events.

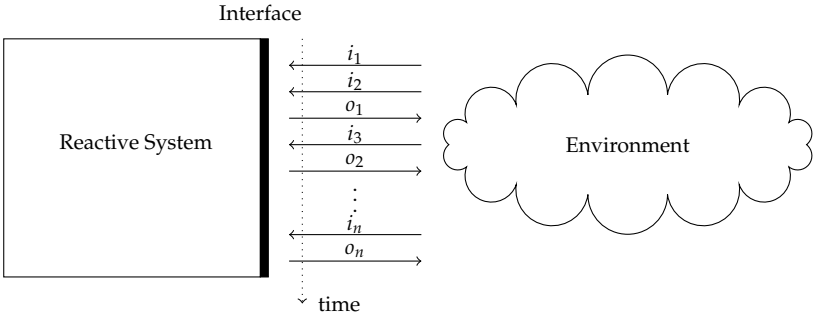


FIGURE 2.1: Reactive systems.

Examples of reactive systems are control programs or operating systems, communication protocols, web-servers, financial services, air-traffic or automobile or nuclear plant control systems and more generally embedded systems which embed software running on hardware devices. Lightweight reactive systems include mobile phones or tablets applications and IoT (Internet of Things) applications composed of small connected devices (sensors, actuators, processors, ...) to the Internet.

2.1.2 Models for reactive systems: Automata/Labeled Transition Systems.

Many state-based formalisms exist to model intended behaviors of reactive systems, in particular we can cite Finite State Machines (FSM) [30] and Labeled Transition Systems (LTS) [78] as representative of such formalisms.

FSM is well known as being useful to model behaviors of reactive systems. In particular, many approaches were studied to test a system modeled by a FSM. In the late 70's, Chow [19] introduced a new technique for testing deterministic FSM. Lately, Tripathy et al. [80] presented an approach for non-deterministic FSM. Then, the FSM were applied recently to generate test sequences for distributed systems [16, 49, 84].

LTS have been successfully used in formal analysis and testing of such systems. LTS are a state-based formalism whose transitions are labeled with inputs, outputs or durations in later extensions to time-constrained behaviors. Durations represent that time elapses between actions (as being either inputs or outputs). The introduction of symbolic abstraction of LTS has been considered more recently with the purpose of compacting the representation of system to master the state-explosion in targeted formal analyses. The first extensions to symbolic handling of data are Symbolic Transition Systems (STS for short) [29, 33]. In the work of Gaston et al. [33], they are called Input Output Symbolic Transition Systems (IOSTS). Timed STSs (TSTS) or Timed IOSTS (TIOSTS) which extend STS with clocks have been defined later in different approaches [1, 4, 5, 74]. In the paper of Andrade et al. [1] and Styp et al [74], models are rather a merge of Timed Automata [66] and STS so that clock values represent convex abstractions, known as zones, while [5] directly uses symbolic techniques to homogeneously handle both clocks and other data variables. Now the expressiveness of such symbolic models is as important as associated underlying exploration and unfolding techniques. For this, STS models are endowed with efficient execution mechanisms using symbolic execution techniques [44] (denoted by SE). The main principle of symbolic execution is to explore systems models for symbolic parameters rather than concrete values and to compute logical constraints on those parameters as the exploration progresses. As glimpsed before, we recall that such technique has been used to handle only data in [1, 74], while it is used to handle both data and time in the work presented by Bannour et al. [4, 5]. The Diversity tool [3] implements SE of TIOSTS (those defined in the papers [5, 33]) and provides them in addition with a textual syntax and editing facilities. We ground the symbolic framework in our work on TIOSTS and their encoding in the Diversity tool. The latter constitutes the tool supporting the symbolic analyses that we propose.

2.2 Timed Input Output Symbolic Transition Systems (TIOSTS)

In this Section, we present Timed Input Output Symbolic Transition Systems (TIOSTS). We illustrate then the usage of the formalism to specify an Automatic Teller Machine (ATM) and a central Bank cooperating together to form a Banking system.

2.2.1 Formalism overview

TIOSTS are defined over a signature $\Sigma = (\Omega, A, K, C)$ where:

- $\Omega = (S, F)$ is an equational logic signature with S a set of names of sorts and F a set of operation names, each with an arity in S. The operations of F are interpreted in the usual way via a model M of a set of concrete values.
- A is a set of variables for storing the received values via the reception of messages, for representing the evolution of the data of the system and finally for defining the guards of the transitions, which constrain their execution.

- K is a set of clocks, which are particular variables whose values belong to a set D_+ (for durations) isomorphic to the set of positive reals, and they are used to denote the time elapsing, in particular durations between two actions. By hypothesis, M contains D_+.
- Finally, C is a set of communication channels partitioned as follows:

$$C = C^{in} \amalg C^{out}$$

where C^{in} is the set of input channels and C^{out} is the set of output channels.

The set of terms $\mathcal{T}_\Omega(A \cup K)$ is inductively defined in the usual way over Ω and $A \cup K$ and the interpretation of variables is canonically extended to terms.

Values of variables of A are updated by using substitutions of the form

$$\rho : A \cup K \to \mathcal{T}_\Omega(A \cup K)$$

We denote $\mathcal{T}_\Omega(A \cup K)^{A \cup K}$ the set of all such substitutions.

In the numerical counterpart, we consider interpretations of the form

$$\nu : A \cup K \to M$$

associating variables with their concrete values in M. We denote $M^{A \cup K}$ the set of all such interpretations.

The set of formulas $\mathcal{F}_\Omega(A \cup K)$ contains the truth values *true* and $false$ and is inductively defined over equality predicates and over usual existential ($\exists$) and universal ($\forall$) quantifiers and boolean connectives ($\vee$, $\wedge$ and $\neg$).

The set of communication actions $Act(\Sigma)$ is defined then as follows:

$$Act(\Sigma) = I(\Sigma) \cup O(\Sigma)$$

where $I(\Sigma) = \{c?x | x \in A, c \in C^{in}\}$ and $O(\Sigma) = \{c!t | t \in T_\Omega(A), c \in C^{out}\}$.

Remark 2.2.1. In the examples we will use a generalized notion of communication actions carrying $n \geq 0$ pieces of data as being obvious extensions of those defined before. That is, an input action can be of the form $c?(x_1, \ldots, x_n)$ or of the simple form $c?$ and an output action can be of the form $c!(t_1, \ldots, t_n)$ or of the simple form $c!$.

A TIOSTS is a triple $\mathbb{G} = (Q, q_0, Tr)$ where:

- Q is a set of control points of the automata, called states,
- q_0 is a distinguished state of Q, called the initial state,
- $Tr \subseteq Q \times 2^K \times \mathcal{F}_\Omega(A \cup K) \times Act(\Sigma) \times \mathcal{T}_\Omega(A \cup K)^A \times Q$ is a set of transitions.

  A transition is a tuple $tr = (q, \mathbb{K}, \phi, act, \rho, q')$ where

  - q (respectively q') is the source (respectively target) state of the transition,
  - $\mathbb{K}$ is a set of clocks to reset,
  - ϕ is a formula which constrains the transition firing,
  - act is a communication action being either an input action or an output action

– and ρ is a substitution of variables (other than clocks) which represents their updates with new values.

Notation 2.2.1. In the sequel, for a given transition $tr = (q, \mathbb{K}, \phi, act, \rho, q')$, we use the notations $src(tr), tgt(tr), Clk(tr), \phi(tr), act(tr), \rho(tr)$ to denote respectively source state q, target state q', set of clocks to reset, formula constraining the transition, action and substitution of data variables of tr.
Moreover, $chan(tr)$ is called the channel of the transition and is used to denote the channel of the action in the transition, i.e. $chan(act(tr))$.

A finite path of a TIOSTS $\mathbb{G} = (Q, q_0, Tr)$ is a sequence of transitions $tr_1 \dots tr_n$ of $\mathbb{G}$ with $tr_i \in Tr$ for $i \leq n$ and such that $tgt(tr_i) = src(tr_{i+1})$ for any $i < n$.
We denote $Paths(\mathbb{G})$ the set of all such paths.

Example 2.2.1 (TIOSTS)**.** Figure 2.2 depicts an excerpt [1] of a TIOSTS representing an Automatic Teller Machine (ATM).

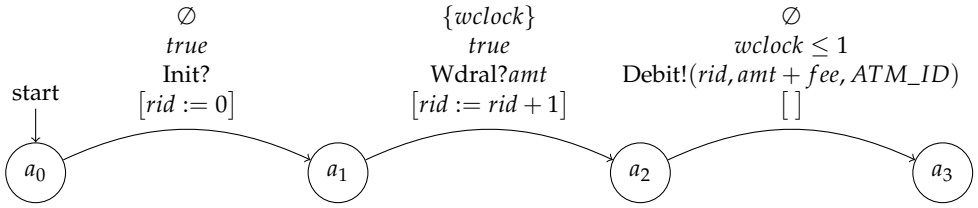


FIGURE 2.2: Excerpt of a TIOSTS for an Automatic Teller Machine.

This simple TIOSTS consists of four states $Q = \{a_0, a_1, a_2, a_3\}$, where a_0 is the initial state, $A = \{rid, amt, fee\}$ is the set of variables, $K = \{wclock\}$ is the set of clocks, $C = \{\text{Init}, \text{Wdral}, \text{Debit}\}$ is the set of channels and $Tr = \{tr_1^a, tr_2^a, tr_3^a\}$ is the set of transitions. Let us discuss the role of each transition:

- Transition $tr_1^a : a_0 \xrightarrow{\emptyset,\ true,\ \text{Init?},\ [rid:=0]} a_1$ [2] represents the initialization transition of the ATM machine which is unconstrained. It is freely fired while the ATM receives a message on the input channel Init from the environment ($act(tr_1^a) = \text{Init?}$). There is no clock to be reset by the transition ($Clk(tr_1^a) = \emptyset$). After the reception, only the value of the variable rid which represents the request identifier is updated (assigned to 0);

- Transition $tr_2^a : a_1 \xrightarrow{\{wclock\},\ true,\ \text{Wdral?}amt,\ [rid:=rid+1]} a_2$ has an input action which represents a reception on the channel Wdral of a withdrawal request. The requested amount is stored in the variable amt. Similar to tr_1^a, the transition is unconstrained that means the transition is freely fired from its source state $src(tr_2^a) = a_1$ and leads to its target state $tgt(tr_2^a) = a_2$. Then, the request identifier rid is incremented. Finally, the transition resets the only clock $wclock$ of ATM;

- Transition $tr_3^a : a_2 \xrightarrow{\emptyset,\ wclock\leq 1,\ \text{Debit!}(rid,amt+fee,ATM\_ID),\ [\ ]} a_3$ illustrates an output transition of ATM to emit a reply on the channel Debit. It is conditioned by the formula $wclock \leq 1$, used to constrain the

[1] Extract of the Banking system will be discussed in Section 2.2.3.

[2] In the following, we use the superscript a and b to denote respectively the transitions of ATM and Bank, such as tr_1^a, tr_2^a (for ATM) and tr_1^b, tr_2^b (for Bank).

duration which has been elapsed from the last transition tr_2^a to tr_3^a. The reply of ATM is a tuple of three elements $(rid, amt + fee, ATM\_ID)$ where rid is the identifier granted with the actual request, $amt + fee$ is the amount which has been debited in the user's account (a fee will be charged when a request is performed) and ATM_ID is a constant (sort *integer*) which represents the ATM identifier in the Banking system. Without loss of generality, we suppose that the value of the constant ATM_ID is 1. Finally, the transition does not modify the values of variables (denoted by $[\,]$).

In the following we give as examples three possible paths of the TIOSTS given above:

- $str_1 = tr_1^a$
- $str_2 = tr_1^a \cdot tr_2^a$
- $str_3 = tr_1^a \cdot tr_2^a \cdot tr_3^a$

The set of paths of this TIOSTS is defined as follows $\{\varepsilon, str_1, str_2, str_3\}$ where ε denotes the empty path.

2.2.2 Semantics of TIOSTS

We show next how to give a semantic counterpart to a TIOSTS in terms of events and timed traces. An execution of a TIOSTS $\mathbb{G} = (Q, q_0, Tr)$ is always described up to a "starting point" which is in fact a semantical denotation of a state of $\mathbb{G}$ called *snapshot*. A snapshot $snp \in Q \times M^{A \cup K}$ is a couple (q, ν) composed of two pieces of information:

- q is a state of $\mathbb{G}$ reached so-far, it conditions which transitions of $\mathbb{G}$ are to be fired next,
- ν is an interpretation of variables of $\mathbb{G}$, it denotes the concrete values which are currently associated with these variables.

The semantics of a TIOSTS are obtained by concretely executing it. The execution of a TIOSTS is based on concrete actions. The set of concrete communication actions $Act(C)$ is defined as follows:

$$Act(C) = I(C) \cup O(C)$$

where $I(C) = \{c?v | v \in M, c \in C^{in}\}$ and $O(C) = \{c!v | v \in M, c \in C^{out}\}$.

A timed trace σ is a sequence $(d_1, act_1) \dots (d_n, act_n)$ where (d_i, act_i) is called an *event* which constitutes a concrete communication action $act_i \in Act(C)$ and $d_i \in D_+$ the time elapsed between the two actions act_{i-1} and act_i (where D^+ is the set of strictly positive real numbers). In the remainder, for an event ev of the form (d, act), we denote $delay(ev)$ the delay d and $act(ev)$ the action act. The set of events is denoted as $Evt(C)$.

The set of runs of a transition $tr \in Tr$, denoted $Run(tr)$, is defined by triples of the form $r = (snp, ev, snp')$ such that if we denote $snp = (q, \nu)$ and $snp' = (q', \nu')$, we have $q = src(tr)$ and the following holds:

- $q' = tgt(tr)$ and the interpretation $\nu' : A \cup K \to M$ is defined as:

$$\nu'(w) = \begin{cases} \nu'_0(\rho(tr)(w)) & \textbf{if } w \in A \\ 0 & \textbf{if } w \in Clk(tr) \\ \nu'_0(w) & \textbf{if } w \in K \setminus Clk(tr) \end{cases} \tag{2.1}$$

where ν'_0 is the auxiliary function defined as:

$$\nu'_0(w) = \begin{cases} v & \textbf{if } act(tr) = c?w \textbf{ with } v \in M \\ \nu(w) & \textbf{if } w \in A \textbf{ with } ((act(tr) = c?x \textbf{ for } w \neq x) \\ & \quad \textbf{or if } act(tr) \neq c?x) \\ \nu(w) + d & \textbf{if } w \in K \textbf{ with } d \in D \end{cases} \tag{2.2}$$

- $delay(ev) = d$
- $act(ev)$ is defined as:

$$act(ev) = \begin{cases} c!\nu'_0(t) & \textbf{if } (act(tr) = c!t) \\ c?\nu'_0(x) & \textbf{if } (act(tr) = c?x) \end{cases} \tag{2.3}$$

- the formula $\phi(tr)$ is satisfiable under the interpretation ν'_0.

Example 2.2.2 (Run of transitions)**.** We may consider $snp_0 = (a_0, [rid := 12, amt := 3, fee := 2, wclock := 16])$ as a possible snapshot from which tr_1^a is a candidate transition to be fired.

Let us consider a snapshot snp_1 which is obtained by executing the transition tr_1^a from snp_0. According to the definition 2.1, we have $\nu'(rid) = \nu'_0(\rho(tr_1^a)(rid))$, and $\rho(tr_1^a)(rid) = 0$, thus $\nu'(rid) = \nu'_0(0)$ and $\nu'(rid) = 0$, since a substitution of an integer number is itself. Similarly, we have $\nu'(amt) := \nu'_0(\rho(tr_1^a)(amt)) = \nu'_0(amt)$, by applying the definition 2.2, we obtain $\nu'_0(amt) = \nu(amt) = 3$, therefore $\nu'(amt) := 3$. In a similar way, $\nu'(fee) := 2$. Moreover, we suppose that $wclock$ reaches 16.4 which means that 0.4 time units has been elapsed from snp_0. The snapshot snp_1 is the following $snp_1 = (a_1, [rid := 0, amt := 3, fee := 2, wclock := 16.4])$. The run of the transition tr_1^a is defined by the triple $r_1 = (snp_0, (0.4, \text{Init?}), snp_1)$.

From snp_1, the TIOSTS can execute the transition tr_2^a. The definition 2.1 gives us the following substitution $\nu'(rid) = \nu'_0(\rho(tr_2^a)(rid)) = \nu'_0(rid + 1) = \nu'_0(rid) + 1$. Moreover, the value of rid is not modified by the input received from the environment, thus $\nu'(rid) = 0 + 1 = 1$. We can easily obtain $\nu'(fee) = 2$. The new value of amt is determined as $\nu'(amt) = \nu'_0(\rho(tr_2^a)(amt)) = \nu'_0(amt)$. Since amt is used to store the reception on the channel Wdral, its value will be modified, let suppose that $amt := 10$ after the reception. We assume also that the time has elapsed 0.5 time units from the last snapshot snp_1. Finally, the clock $wclock$ is reset. The snapshot snp_2 is defined as follows $snp_2 = (a_2, [rid := 1, amt := 10, fee := 2, wclock := 0])$. The run of the transition tr_2^a is defined by the triple $r_2 = (snp_1, (0.5, \text{Wdral?}10), snp_2)$.

Let us consider the last example of runs by executing the transition tr_3^a from snp_2. By using the definition 2.1 and 2.2, we obtain $\nu'(rid) = \nu'_0(\rho(tr_3^a)(rid)) = \nu'_0(rid) = \nu(rid) = 1$. Similarly, we have $\nu'(fee) = 2$, $\nu'(amt) = 10$. The transition tr_2^a is conditioned by the guard $wclock \leq 1$ which permits it to be fired within at most 1 time unit from the last run. We suppose that $wclock$ has elapsed 1. The snapshot snp_3 is the following $snp_3 = (a_3, [rid := 1, amt := 10, fee := 2, wclock := 1])$ and the run of tr_3^a is the triple $r_3 = (snp_2, (1, \text{Debit!}(1, 12, 1)), snp_3)$.

Notation 2.2.2. In the sequel, for a given run $r = (snp, ev, snp')$, we use the notations $src(r)$, $ev(r)$ and $tgt(r)$ to denote respectively snp, ev and snp'.

For any sequence of runs $rseq = r_1 \dots r_n$, the trace of $rseq$ is $trace(rseq) = ev(r_1) \dots ev(r_n)$. For any path $p = tr_1 \dots tr_n \in Paths(\mathbb{G})$, a run of p is defined as $r_1 \dots r_n$ such that $r_i \in Run(tr_i)$, $tgt(r_i) = src(r_{i+1})$ and $\nu(src(r_1))(w) = 0$ for any $w \in K$. We denote $Run(p)$ the set of all such runs.

The set of runs of $\mathbb{G}$ is the set

$$Run(\mathbb{G}) = \bigcup_{p \in Paths(\mathbb{G})} Run(p)$$

The set of traces of $\mathbb{G}$ is the set:

$$Trace(\mathbb{G}) = \bigcup_{rseq \in Run(\mathbb{G})} trace(rseq)$$

Example 2.2.3 (Traces)**.** As an illustration, we introduce several examples of traces.

- For the sequence of runs $rseq_1 = r_1 \cdot r_2$, we obtain the trace

$$trace(rseq_1) = (0.4, \text{Init?}) \cdot (0.5, \text{Wdral?}10)$$

- For the sequence of runs $rseq_2 = r_1 \cdot r_2 \cdot r_3$, we obtain the trace

$$trace(rseq_2) = (0.4, \text{Init?}) \cdot (0.5, \text{Wdral?}10) \cdot (1, \text{Debit!}(1, 12, 1))$$

A TIOSTS is *deterministic* if and only if for any $p \in Paths(\mathbb{G})$ and for any two transitions $tr_1, tr_2 \in Tr$ such that $p \cdot tr_1$, $p \cdot tr_2 \in Paths(\mathbb{G})$, we have that for any $rseq \in Run(p)$ and for any run $r_1 = Run(tr_1)$ and $r_2 = Run(tr_2)$ the following holds:

$$rseq.r_1 \in Run(\mathbb{G}) \wedge rseq.r_2 \in Run(\mathbb{G}) \implies ev(r_1) \neq ev(r_2)$$

In other words, a TIOSTS is deterministic if for any trace, there is exactly one path of the TIOSTS giving rise to this trace.

Example 2.2.4 (Deterministic TIOSTS)**.** In each following example of TIOSTS, amt and fee are natural numbers.

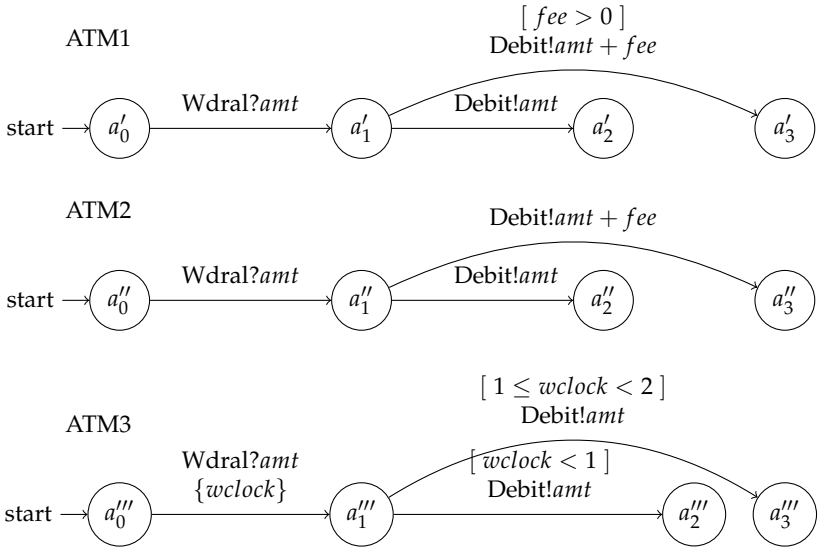


FIGURE 2.3: Examples of ATM

Example of ATM1.

Let suppose that we have a sequence of runs $rseq = r_1$ with $r_1 \in Run(tr_1)$ (tr_1 denotes the transition $a'_0 \rightarrow a'_1$) and two runs $r_2 \in Run(tr_2)$ and $r_3 \in Run(tr_3)$ (tr_2 and tr_3 denote respectively the transitions $a'_1 \rightarrow a'_2$ and $a'_1 \rightarrow a'_3$). Without loss of generality, we assume that $ev(r_1) = (4, \text{Wdral?}5)$, that means after the execution of r_1, amt is assigned to the value of 5, since the value of fee is always constrained by the condition $fee > 0$ in tr_3, consequently $5 + fee > 5$, i.e. there does not exist r_2 and r_3 such that $ev(r_2) = ev(r_3)$. Thus, ATM1 is deterministic.

Example of ATM2.

We consider the sequences of run r_1 with $r_1 \in Run(tr_1)$ (tr_1 denotes the transition $a''_0 \rightarrow a''_1$). Once again, we assume that $ev(r_1) = (4, \text{Wdral?}5)$. In case of $fee := 0$, we can always find r_2 and r_3 with $r_2 \in Run(tr_2)$ and $r_3 \in Run(tr_3)$, such that $ev(r_2) = ev(r_3)$ (since $amt + fee = amt$ for $fee = 0$). Therefore, ATM2 is a non-deterministic TIOSTS.

Example of ATM3.

Similarly, let consider two sequences of runs denoted by $r_1 \cdot r_2$ and $r_1 \cdot r_3$ with $r_1 \in Run(tr_1), r_2 \in Run(tr_2), r_3 \in Run(tr_3)$. It is not possible for ATM3 to have $ev(r_2) = ev(r_3)$ since $delay(ev(r_2))$ is constrained by the condition $1 \leq delay(ev(r_2)) < 2$ and $delay(ev(r_3))$ is constrained by $delay(ev(r_2)) < 1$. By consequence, ATM3 is deterministic.

2.2.3 Running case study: ATM-Bank system

In order to demonstrate the usage of TIOSTS for modeling behavior of reactive systems, we introduce an illustrative example called banking system where several ATM (stands for Automatic Teller Machine [3]) exchange information with different bank agencies. In particular, we develop respectively two TIOSTS models representing the generic behavior of an ATM (Figure 2.4) and a bank agency (Figure 2.5).

2.2.3.1 TIOSTS of ATM

Let us describe in the following behaviors of the TIOSTS ATM. For the sake of clarity, we only present a simplified view of the system ATM.

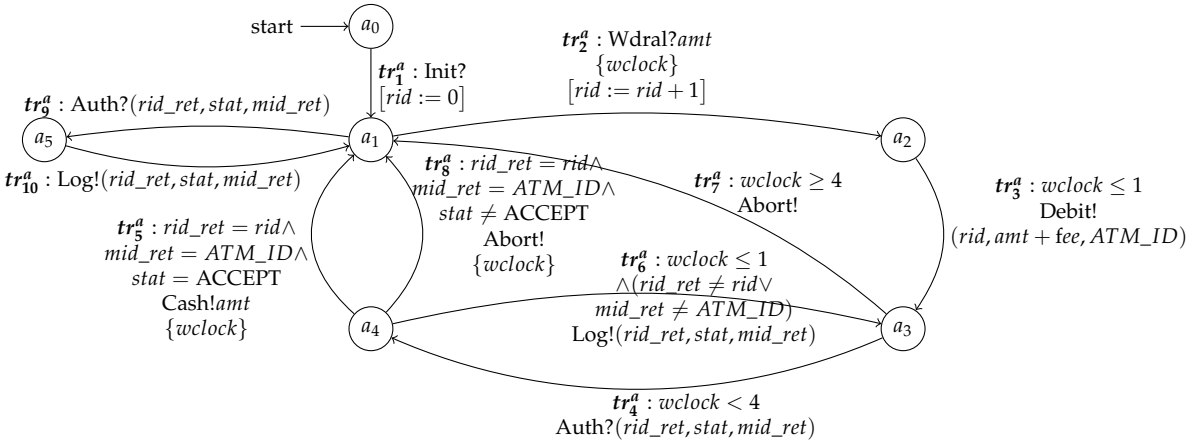


FIGURE 2.4: TIOSTS of ATM

[3] An automated teller machine (ATM) is an electronic telecommunications device that allows customers to perform financial transactions, such as cash withdrawals, deposits, transfer funds, or obtaining account information, at any time and without the need for direct interaction with bank staff.

An ATM is described by the TIOSTS given in Figure 2.4. Let us provide in the following the complete signature of ATM. This TIOSTS is defined over the signature $\Sigma_a = (\Omega, A_a, K_a, C_a)$ where:

- $A_a = \{rid, amt, fee, rid\_ret, stat, mid\_ret\}$ is the set of variables,
- $K_a = \{wclock\}$ is the set of clocks,
- $C_a = \{\text{Init}, \text{Wdral}, \text{Debit}, \text{Auth}, \text{Abort}, \text{Log}, \text{Cash}\}$ is the set of channels.

In the following, we introduce the role of each transition in the TIOSTS ATM. tr_1^a, tr_2^a, tr_3^a representing respectively transition $a_0 \rightarrow a_1, a_1 \rightarrow a_2, a_2 \rightarrow a_3$ will not be discussed in this section since they are all previously presented in the example 2.2.1. We remark that ATM_ID is a constant of the TIOSTS ATM. Without loss of generality, we suppose that the ATM_ID is predefined as 1.

- $tr_4^a : a_3 \xrightarrow{\emptyset, wclock<4, \text{Auth?}(rid\_ret, stat, mid\_ret), [\,]} a_4$ represents the response that is received by the ATM within the specified duration. The new values of the reception are then stored in the variables $rid\_ret, stat, mid\_ret$. This response is then analyzed to provide a final reply to the client;
- $tr_5^a : a_4 \xrightarrow{\{wclock\}, rid\_ret=rid \wedge mid\_ret=ATM\_ID \wedge stat=ACCEPT, \text{Cash!}amt, [\,]} a_1$ returns cash to the client if the request has been accepted;
- $tr_6^a : a_4 \xrightarrow{\emptyset, (wclock \leq 1) \wedge (rid\_ret \neq rid \vee mid\_ret \neq ATM\_ID), \text{Log!}(rid\_ret, stat, mid\_ret), [\,]} a_3$ sends a log message concerning a late answer to some request or an answer containing an incorrect identifier for the ATM. A late answer is simply recognized by an identifier which is different from the current rid;
- $tr_7^a : a_3 \xrightarrow{\emptyset, wclock \geq 4, \text{Abort!}, [\,]} a_1$ illustrates a request cancellation. After transferring the withdrawal request on the channel Debit, the ATM waits 4 time units for the response. Without receiving any response within this duration, the request is aborted;
- $tr_8^a : a_4 \xrightarrow{\{wclock\}, rid\_ret=rid \wedge mid\_ret=ATM\_ID \wedge stat \neq ACCEPT, \text{Abort!}, [\,]} a_1$ is an aborting reply since the request has not been accepted, i.e. value which has been stored in the variable $stat$ is different from $ACCEPT$;
- $tr_9^a : a_1 \xrightarrow{\emptyset, true, \text{Auth?}(rid\_ret, stat, mid\_ret), [\,]} a_5$ represents a reception of *a non-involved request* from a_1. A non-involved request is defined as the one which has been performed by another client (not the actual one) who abandoned the request for some reason;
- $tr_{10}^a : a_5 \xrightarrow{\emptyset, true, \text{Log!}(rid\_ret, stat, mid\_ret), [\,]} a_1$ returns a log corresponding to the non-involved request to the bank.

2.2.3.2 TIOSTS of Bank

The TIOSTS of a bank is described in Figure 2.5. In order to manage the withdrawal sum during 3 consecutive days of a client, the TIOSTS uses a circular array of size 3 (i.e. sum) to store withdrawal amounts of the last three days. In practice, the stored amounts for the 3^{th} previous day are not relevant any more. The counter $days$ is incremented at the end of each day and is used to identify the next place in the array sum to store the accumulated amounts for the new day by using the operator modulo. In the following, we discuss the different roles of the bank. The first one is to assign authorisations to withdrawal requests:

- $tr_1^b : b_0 \xrightarrow{\{tclock,dclock,pclock\},true,\text{Init!},\left[days:=0,sum:=\{0,0,0\},cpt:=0,tloc:=0\right]} b_1$ sends an initialization message to all of the ATM machine it manages;
- $tr_2^b : b_1 \xrightarrow{\{pclock\},dclock+6<24\text{H},\text{Debit?}(tid,x,aid),\left[\ \right]} b_2$ receives a new request on the channel Debit within a day. Consequently, a new process is instantiated. The duration of each process is measured by the clock *pclock*. We remark that each process is constrained to terminate within the current day, therefore we add a small duration (6 time units) to our constraints such that the bank has enough time to react (e.g. $dclock + 6 < 24\text{H}$). There are three possible processes that will be discussed;
- $tr_3^b : b_2 \xrightarrow{\{pclock\},tclock\geq tloc\wedge pclock\leq 6\ldots,\text{Auth!}(tid,\text{ACCEPT},aid),\left[bal=-x,cpt++,\ldots\right]} b_1$. The request is accepted if the balance is sufficient and the total debit limit over a rolling 3 days is not exceeded (the lowest transition $b_2 \rightarrow b_1$), this leads to the update of the withdrawal amount by accumulating the requested amount x and the update of balance by deducting the amount x;
- $tr_4^b : b_2 \xrightarrow{\{pclock\},tclock<tloc\wedge pclock\leq 6\ldots,\text{Auth!}(tid,\text{LOCK},aid),\left[\ \right]} b_1$. The request is locked if the bank receives it during the lock time;
- $tr_5^b : b_2 \xrightarrow{\{pclock\},pclock\leq 6\wedge(x>bal\vee x<100\vee(x+\sum_{i\leq 3} sum[i]>1000)),\text{Auth!}(tid,\text{REJECT},aid),\left[\ \right]} b_1$. The request is rejected if either the total debit limit is exceeded or the requested amount is greater than the balance or less than 100 money units;

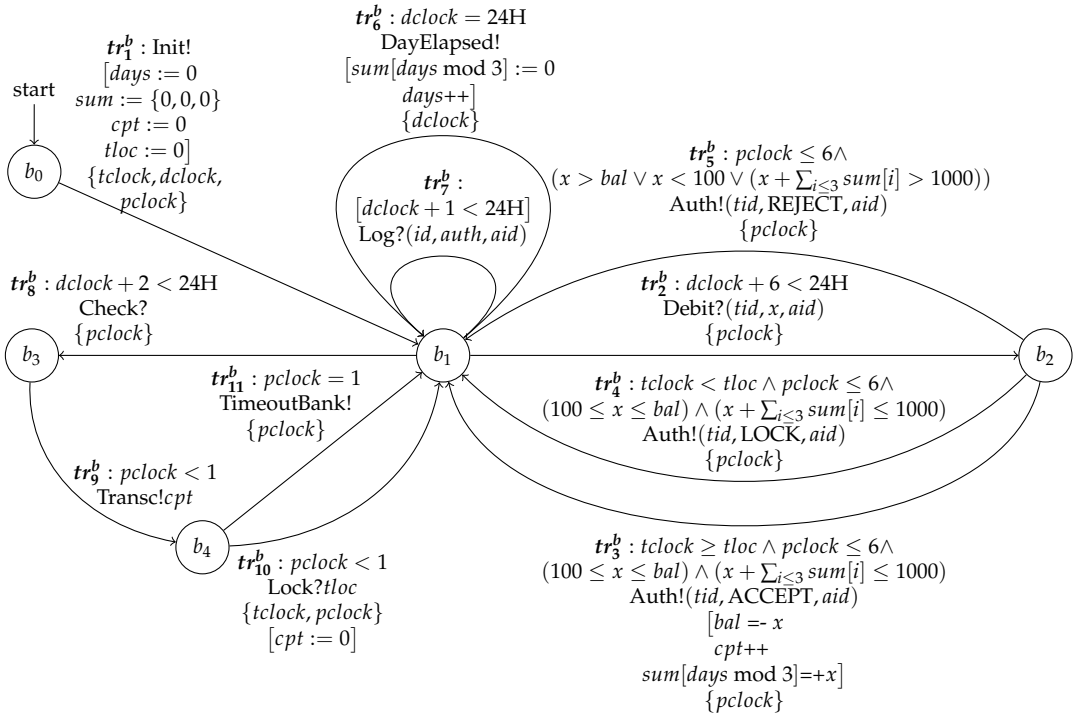


FIGURE 2.5: TIOSTS of Bank

The second functionality of the Bank is to elapse time during the current day:

- $tr_6^b : b_1 \xrightarrow{\{dclock\},dclock{=}24H,\text{DayElapsed!},\left[sum[days \bmod 3]:=0,days{+}{+}\right]} b_1$. When a day has elapsed (i.e. $dclock = 24H$), the counter $days$ is incremented and the withdrawal amount for the new day is reset to zero;

- $tr_7^b : b_1 \xrightarrow{\emptyset,dclock{+}1{<}24H,\text{Log?}(id,auth,aid),[\,]} b_1$. During the day, the bank can receive at any moment a log message from an ATM.

Finally, the bank checks the number of successful requests from a client within the current day:

- $tr_8^b : b_1 \xrightarrow{\{pclock\},dclock{+}2{<}24H,\text{Check?},[\,]} b_3$ allows a banker to check the number of successful requests within a day;

- $tr_9^b : b_3 \xrightarrow{\emptyset,pclock{<}1,\text{Transc!}cpt,[\,]} b_4$ emits a reply to the banker in the following time unit;

- $tr_{10}^b : b_4 \xrightarrow{\{tclock,pclock\},pclock{<}1,\text{Lock?}tloc,\left[cpt:=0\right]} b_1$. The banker has less than one time unit to update the lock time if he finds that there are too many transactions during some period. Otherwise the time is expired $tr_{11}^b : b_4 \xrightarrow{\{pclock\},pclock{=}1,\text{TimeoutBank!},[\,]} b_1$.

2.3 Symbolic execution of TIOSTS

The symbolic execution of TIOSTS has been defined in [4, 5, 32]. It boils down to the construction of a *symbolic tree* obtained by *symbolically executing* transitions not for concrete values but rather using fresh variables (or symbolic parameters). For this, a set of fresh variables F disjoint from TIOSTS variables is considered, that is $F \cap (A \cup K) = \emptyset$. We denote $F_t \subseteq F$ the set of fresh variables of type time which is used to symbolically denote durations associated with transitions execution. Similarly, $F_d \subseteq F$ is the set of fresh variables used for variables of other types. For the signature $\Sigma_F = (\Omega, F, \emptyset, C)$, the set of *symbolic events* over Σ_F is

$$Evt(\Sigma_F) ::= F_t \times Act(\Sigma_F)$$

Notation 2.3.1. For any event $ev = (z, act)$ in $Evt(\Sigma_F)$, we denote $delay(ev)$ and $act(ev)$ respectively z and act.

2.3.1 Principle of symbolic execution

The symbolic execution technique is based on an execution context, abbreviated by EC. An execution context $ec = (q, \pi, \lambda, ev, pec)$ is a data structure which is composed of the following pieces of information:

- a state (control point) q of TIOSTS which is reached by the execution so far, it is a source state from which candidate transitions can be be executed,

- a path condition π, the so-called path condition, which is a formula representing all conditions over variables and clocks that need to be satisfiable by the symbolic execution to reach ec,

- a substitution of variables λ of the TIOSTS by expressions over fresh variables, that denote their current associated values,

- a symbolic event ev which is the event that has been executed to reach ec,
- and finally, a predecessor of ec, denoted by pec, that represents the execution context from which ec has been created by executing the corresponding transition. For the initial execution context, this component is left undefined.

While components q, π and λ of an EC suffice to fully characterize by means of symbolic variables the current state (control state and possible values for state variables), components ev and pec of an EC are primarily useful to build a tree in which nodes are ECs and edges connect predecessor ECs to ECs themselves.

An execution context has one and only one predecessor, whereas it can have many successors since from a state in a TIOSTS, we can execute more than one candidate transition. Therefore, from an execution context, we can have access to its predecessor as well as its pieces of information.

Notation 2.3.2. For a given execution context ec, we use respectively the notations $q(ec), \pi(ec), \lambda(ec), ev(ec)$ and $pec(ec)$ to denote the corresponding elements in ec.

For a TIOSTS $\mathbb{G} = (Q, q_0, Tr)$, symbolically executing a transition $tr \in Tr$ from an EC $ec = (q, \pi, \lambda, ev, pec)$ boils down to determination of the successor EC ec' which is reached from ec by taking into account the elements of tr. If we note $tr = (q, \mathbb{K}, \phi, act, \rho, q') \in Tr$, the EC $ec' = (q', \pi', \lambda', ev', pec')$ is defined as follows:

- the substitution $\lambda' : A \cup K \to \mathcal{T}_\Omega(F)$:

$$\lambda'(w) = \begin{cases} \lambda'_0(\rho(w)) & \textbf{if } w \in A \\ 0 & \textbf{if } w \in \mathbb{K} \\ \lambda'_0(w) & \textbf{if } w \in K \setminus \mathbb{K} \end{cases} \tag{2.4}$$

where λ'_0 is the auxiliary function defined as:

$$\lambda'_0(w) = \begin{cases} y & \textbf{if } act = c?w \textbf{ with } y \in F_{type(w)} \\ \lambda(w) & \textbf{if } w \in A \textbf{ with } act = c?x \textbf{ for } w \neq x \\ & \textbf{or with } act \neq c?x \\ \lambda(w) + z_0 & \textbf{if } w \in K \textbf{ with } z_0 \in F_t \end{cases} \tag{2.5}$$

- the guard π' is $\pi \wedge \lambda'_0(\phi)$.
- $delay(ev') = z_0$ with $z_0 \in F_t$ which is precisely the variable used in the definition of λ'_0 and:

$$act(ev') = \begin{cases} c!\lambda'_0(t) & \textbf{if } act = c!t \\ c?\lambda'_0(x) & \textbf{if } act = c?x \end{cases} \tag{2.6}$$

- and finally $pec' = ec$.

The symbolic execution of a transition translates directly the standard execution of the involved transition: λ'_0 is an intermediate substitution which advances for all clocks a same symbolic duration z_0 to denote time passing, assigns to a data variable w a fresh variable y if w is the variable of a reception $(c?w)$ and leaves unchanged other data variables. The substitution λ' for w is obtained after the application of λ'_0 on the substitution ρ, clocks that are reset by the transition (determined by the set of clocks $\mathbb{K}$) are reset to 0 and the other clocks advance z_0. The path condition π' is obtained by the accumulation of the predecessor one π and the guard of the transition ϕ evaluated with input fresh variables and time elapsing.

In the following, let us denote $\mathbb{EC}(\mathbb{G})$ the set of all execution contexts of the TIOSTS $\mathbb{G}$. As we can state, from the given context $ec \in \mathbb{EC}(\mathbb{G})$, one can determine the context ec' of $\mathbb{EC}(\mathbb{G})$ based on the previous definition. It remains to make clear what the initial contexts are: they are of the form $(q_0, true, \lambda_0, (0, \_), self)$ with

- the symbol _ for indicating the absence of an action,
- with $\lambda_0 : A \cup K \to \mathcal{T}_\Omega(F)$ associating to variables of A a fresh variable[4] and to variables of K the constant 0,
- $self$ an identifier indicating that the initial context has no predecessor context, except by convention the initial context itself.

Notation 2.3.3. For a given execution context ec, we use the notation $tr(ev(ec))$ to denote the corresponding transition in the TIOSTS $\mathbb{G}$ which has been executed from $pec(ec)$ to obtain ec. Moreover, for simplicity, $act(ec)$, $delay(ec)$ and $chan(ec)$ denotes respectively $act(ev(ec))$, $delay(ev(ec))$ and $chan(act(ev(ec)))$.

Example 2.3.1 (Symbolic execution of TIOSTS)**.** We consider the excerpt ATM in example 2.2.1 and illustrate the symbolic execution through its three transitions.

Given the initial execution context $ec_1 = (a_0, true, \{rid := rid_0, amt := amt_0, fee := fee_0, wclock := 0\}, (0, \_), self)$ where the notation _ denotes the absence of a symbolic action and $self$ denotes that the predecessor of the initial context ec_1 is itself.

The symbolic execution of the transition tr_1^a from ec_1 leads to a new execution context denoted by ec_2. We suppose that the time has elapsed a duration z_0 from ec_1 (z_0 is a new fresh variable). The substitution of λ' applied to rid is the following $\lambda'(rid) = \lambda'_0(\rho(rid)) = \lambda'_0(0) = 0$ (definition 2.4). Similarly, $\lambda'(amt) = \lambda'_0(\rho(amt)) = \lambda'_0(amt)$ (definition 2.4). In addition, $\lambda'_0(amt) = \lambda(amt) = amt_0$ (definition 2.5). The delay $delay(ec_2)$ and symbolic action $act(ec_2)$ are respectively defined as z_0 and Init?. In Figure 2.6, we illustrate the symbolic execution of tr_1 from ec_1 to obtain ec_2. We note also that for the sake of simplicity, only variables whose value has been modified through the symbolic execution are illustrated.

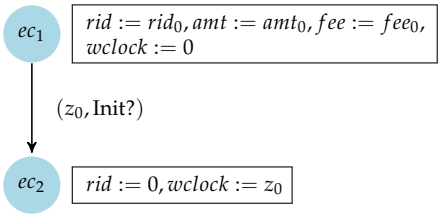


FIGURE 2.6: Symbolic execution of initialization transition

Let us consider the second transition tr_2^a which could be fired from ec_2 and its execution leads to the creation of a new context denoted by ec_3. We denote by z_1 the delay that has been elapsed until the reception

[4] All introduced fresh variables are distinct

on the channel Wdral. The new value of amt is determined as $\lambda'(amt) = \lambda'_0(\rho(amt)) = \lambda'_0(amt)$ (definition 2.2). According to the definition 2.5, $\lambda'_0(amt) = amt_1$ to express that the ATM receives a new value amt_1 which is stored in amt. Similarly, we obtain that $\lambda'(rid) = 1$ and $\lambda'(wclock) = 0$ ($wclock$ is reset). The delay $delay(ec_3)$ and symbolic action $act(ec_3)$ are respectively defined as z_1 and Wdral?amt_1. The path condition $\pi(ec_3)$ is $\pi(ec_2) \wedge \lambda'_0(\phi(tr_2))$, i.e. $\pi(ec_2) \wedge true$, since tr_2^a could be freely fired from a_1.

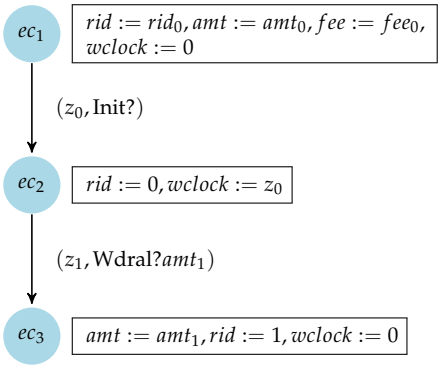


FIGURE 2.7: Symbolic execution of withdrawal transition

From ec_3, the transition tr_3^a could be executed in order to create a new execution context ec_4. The delay of transition is denoted by z_2. All variables remain unchanged since there is no update concerning variables, except that the clock $wclock$ has elapsed z_2 time units from ec_3. The delay $delay(ec_4)$ and symbolic action $act(ec_4)$ are defined as z_2 and Debit!$(1, amt_1 + fee_0, 1)$. This transition is conditioned by the guard $wclock \leq 1$, therefore $\pi(ec_4) = \pi(ec_3) \wedge \lambda'_0(\phi(tr_3^a)) = \lambda'_0(wclock \leq 1) = z_2 \leq 1$.

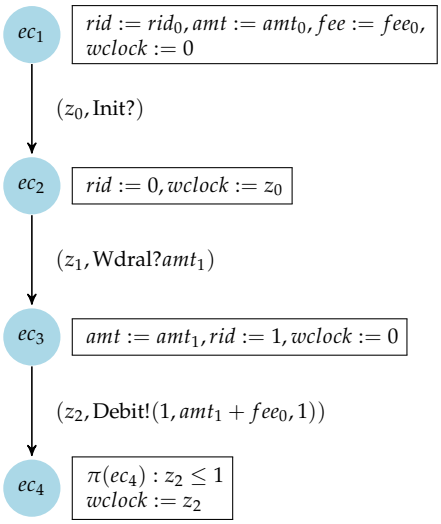


FIGURE 2.8: Symbolic execution of debiting transition

The symbolic execution consists in constructing *symbolic paths* corresponding to finite paths of a TIOSTS. Indeed, for a given finite path $p = tr_1 \dots tr_n$ of a TIOSTS $\mathbb{G}$, its corresponding symbolic path is of the form $sp = ec_1 \dots ec_{n+1}$ where ec_1 is the initial execution context, ec_{i+1} is the context resulting from the symbolic execution of tr_i from the predecessor context ec_i, with $1 \leq i \leq n$. Formally, we have $pec(ec_{i+1}) = ec_i$, with $1 \leq i \leq n$. We denote $SPaths(\mathbb{G})$ the set of all such paths. In the following, for a given symbolic path

$sp = ec_1 \dots ec_{n+1} \in SPaths(\mathbb{G})$, we use the notation $\pi(sp)$ to denote the path condition $\pi(ec_{n+1})$.

As we can see from the example 2.3.1, the symbolic execution of the excerpt ATM is considered as the symbolic execution of all transitions in the TIOSTS from a given EC *ec*. In the following, we denote by $SE(\mathbb{G}) = (ec_1, \mathbb{EC})$ the symbolic execution of a TIOSTS $\mathbb{G}$ where:

- ec_1 is an arbitrary initial execution context,
- $\mathbb{EC}$ is the set of all execution contexts of $\mathbb{G}$.

$SE(\mathbb{G})$ is a tree-like structure whose root is ec_1 and nodes are execution contexts containing pieces of information related to the possible executions of $\mathbb{G}$.

2.3.2 Illustration with ATM

Until now, we have presented the symbolic execution technique and its principle in our approach to obtain the semantics of a TIOSTS. In this section, we illustrate this technique through the examples of TIOSTS ATM. These illustrations show that our work has been not only conceived and implemented in a formal analysis tool, but also it is made to deal with quite complex system models.

Before discussing the illustrations, we take a look at the architecture of the ATM with different communication channels in the figure below. We remark that in this illustration, we consider only the unitary system ATM, therefore all of its communication channels are connected to the environment. Figure 2.9 represents the connection between ATM and the environment (denoted by ENV) through communication channels.

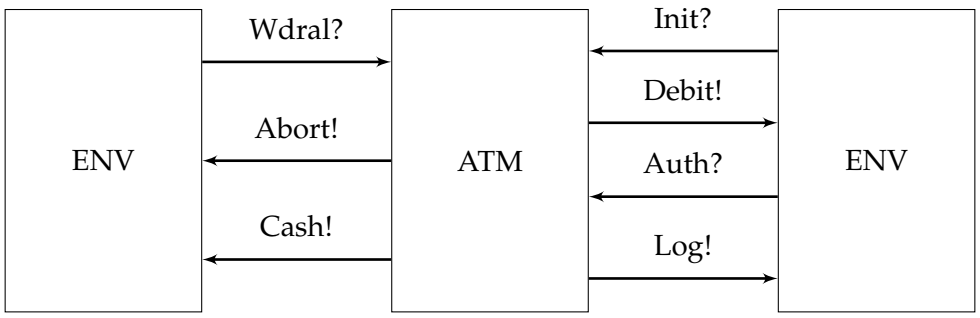


FIGURE 2.9: Connection between ATM and the environment

In the following, let go into details about each execution context which is created by symbolic execution. We suppose that the contexts ec_1, ec_2, ec_3 and ec_4 are similarly computed as in example 2.3.1. As pointed out previously, in the sequel, for the sake of simplicity, the update of data variables and clocks in an execution context is explicitly depicted only when their values have been modified with respect to its predecessor.

From the context ec_4, ATM executes the transition tr_4^a. This transition represents a response that is received by the ATM, new fresh variables $rid\_ret_2, stat_2, mid\_ret_2$ are created to store new values. We denote by z_3 the time elapsed from ec_4 to the reception on the channel Auth, thus the value of clock *wclock* is defined by $z_2 + z_3$. The transition is also conditioned by the guard $wclock < 4$, which implies $z_2 + z_3 < 4$. The path condition $\pi(ec_5)$ is obtained by conjoining $\pi(ec_4)$ and the evaluation of guard of actual transition, $\pi(ec_5) = z_2 + z_3 < 4 \wedge z_2 \leq 1$.

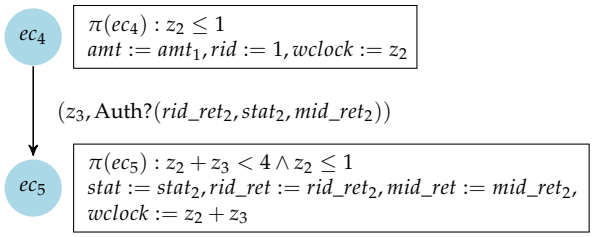


FIGURE 2.10: Symbolic execution of authorization request transition

Next, we consider the output transition tr_8^a. We suppose that the time has elapsed z_4. The variables remain unchanged since there is no update concerning them, except that the clock is reset by the transition, i.e. $wclock := 0$. The transition tr_8^a is conditioned by the following guard over the values which have been received from the previous reception on the channel Auth: $rid\_ret = rid \wedge mid\_ret = ATM\_ID \wedge stat \neq$ ACCEPT, therefore the path condition is constructed as $\pi(ec_7) = \pi(ec_5) \wedge mid\_ret_2 = 1 \wedge rid\_ret_2 = 1 \wedge stat_2 \neq ACCEPT$.

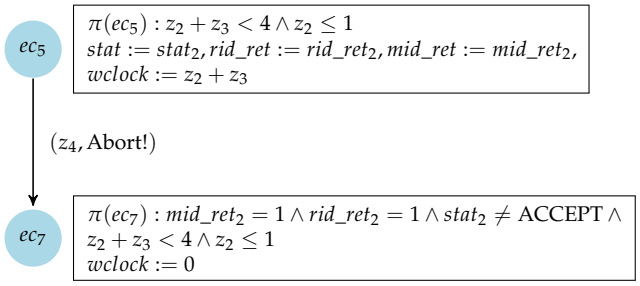


FIGURE 2.11: Symbolic execution of aborting request transition

From the context ec_5, the transition tr_6^a could be executed in order to reach a new execution context ec_8. Similarly, the delay of this transition is denoted by z_4 and the value of clock $wclock$ is $z_2 + z_3 + z_4$. The path condition $\pi(ec_8)$ is defined as $\pi(ec_5) \wedge (rid\_ret_2 \neq 1 \vee mid\_ret_2 \neq 1) \wedge (z_2 + z_3 + z_4 \leq 1)$.

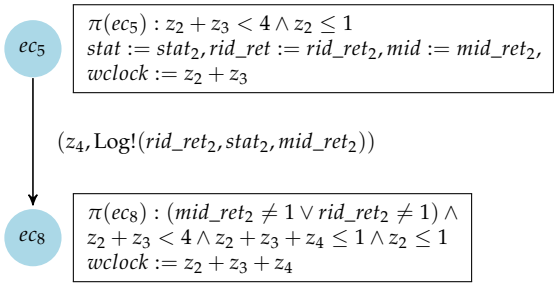


FIGURE 2.12: Symbolic execution of locking request transition

We remark that the symbolic execution in the TIOSTS ATM is a non-stopping process because it is always possible to reach the state a_1 in order to start a new withdrawal request. In Figure 2.13, we present a part

of symbolic execution of ATM which allows to cover each transition in ATM at least once. Let us point out that in the symbolic execution tree, for any two different execution contexts sharing a same predecessor, they are always associated with the same fresh delay. The important thing is that on the same path, each newly introduced fresh variable is different from the ones occurring before in the symbolic path under construction.

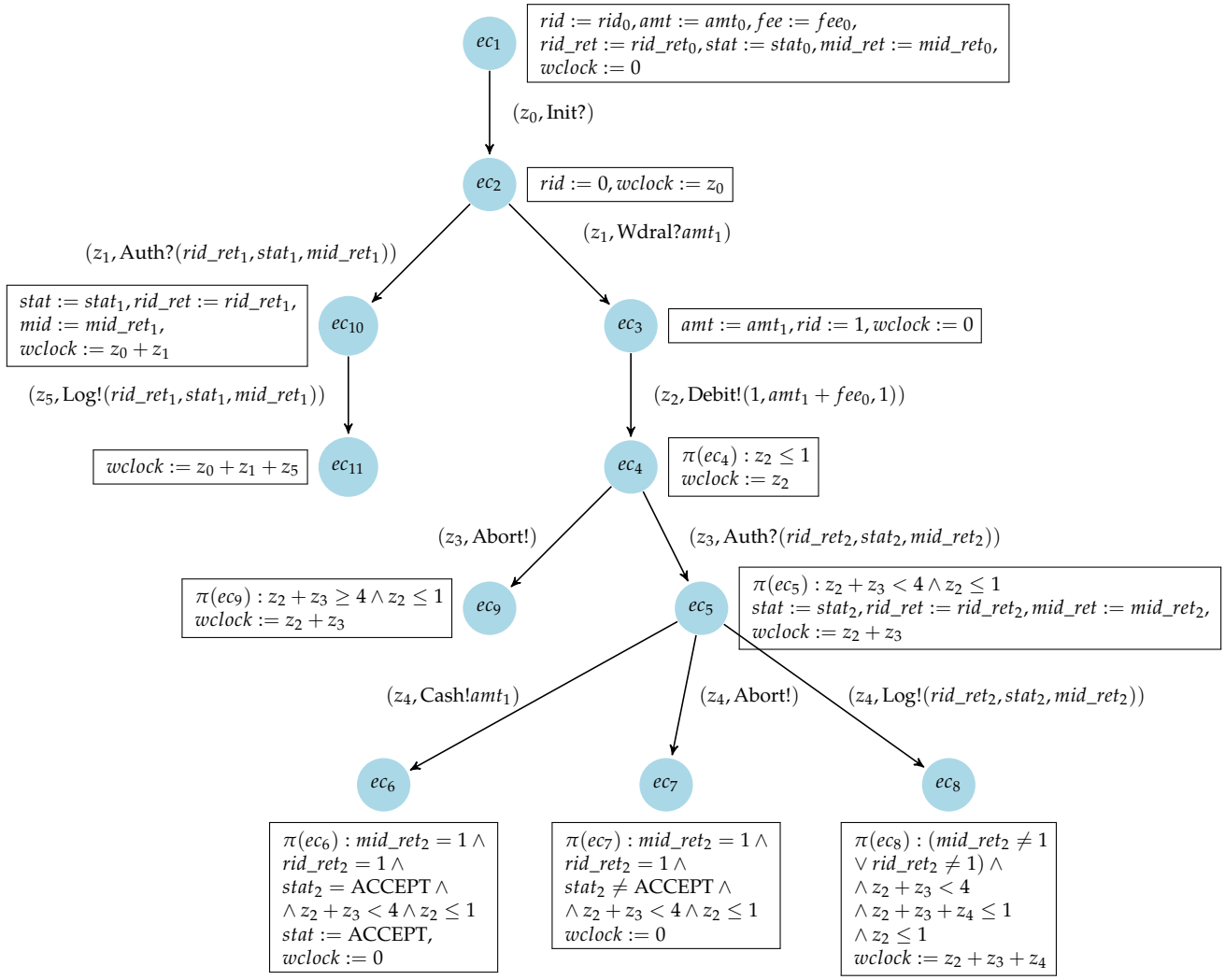


FIGURE 2.13: Symbolic execution of ATM

2.3.3 Deterministic TIOSTS

In this section, we are going to discuss the *determinism* of a TIOSTS. In some existing approaches [1, 42], the authors did not deal with the non-deterministic situations while modeling systems with specific models.

Remark 2.3.1. In the sequel, we suppose that a TIOSTS $\mathbb{G}$ is given. Let consider two symbolic paths of $\mathbb{G}$ (defined in Section 2.3.1) denoted by $p_1 = ec_1 \dots ec_n.ec_{n+1}$ and $p_2 = ec_1 \dots ec_n.ec_{n+2}$ which share a same prefix. Before discussing our proposition for deterministic TIOSTS, we introduce the following restrictions:

- $delay(ec_{n+1}) = delay(ec_{n+2})$, i.e. any two outgoing symbolic transitions from the same execution context are associated with the same fresh duration,

- for any channel c, if $act(ec_{n+1}) = c?x_1$ and $act(ec_{n+2}) = c?x_2$ then $x_1 = x_2$, i.e., any two outgoing symbolic transitions from the same execution context are associated with the same reception variable for a given channel,
- for any symbolic path p, the identification condition of p's observable values is inductively defined as follows:

$$OCond(p) = \begin{cases} true & \textbf{if } p = \epsilon \\ OCond(p') \wedge (\chi = t) & \textbf{if } p = p'.ec \wedge act(ec) = c!t \\ OCond(p') & \textbf{if } p = p'.ec \wedge (act(ec) = c?x \vee act(ec) = \_) \end{cases}$$

  in which $OFresh(pec(ec))_c = \chi$ is a fresh variable [5],

  for any ec, we denote $OCond(ec)$ the identification condition $OCond(p.ec)$ where p is the path that leads to ec, by convention we consider that the path that leads to the initial context ec_1 is the empty path.
- $IFresh(ec_1)$ denotes the set of variables which have been initialized at the beginning of symbolic execution.

Notation 2.3.4. In the following, we use the syntax $\exists_E$, where $E = \{x_1, \ldots, x_n\}$ is a set of variables, to denote the application of existential quantifier to all variables in E, i.e. $\exists x_1, \ldots, \exists x_n$.

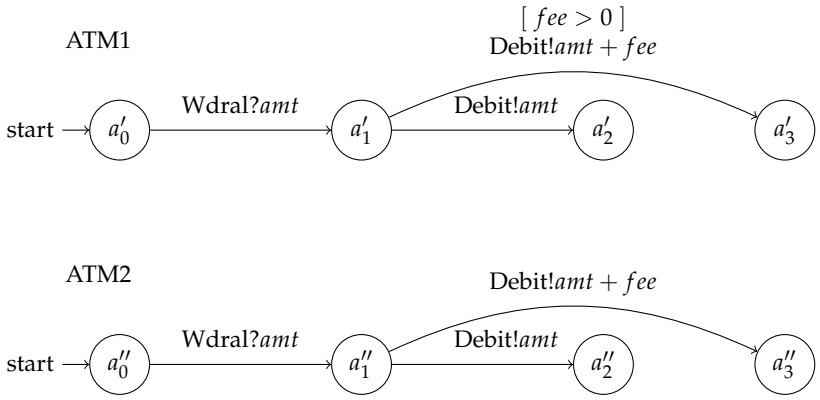
In our work, we will consider in the following only a subclass of TIOSTS called deterministic TIOSTS by making an assumption about the symbolic execution.

Definition 2.3.1 (Deterministic TIOSTS)**.** A TIOSTS $\mathbb{G}$ is deterministic if for any couple of symbolic path $p.ec, p.ec'$ where p is a symbolic path and ec, ec' are execution contexts such that $ec \neq ec'$, $pec(ec) = pec(ec')$ and $chan(ec) = chan(ec')$ we have that:

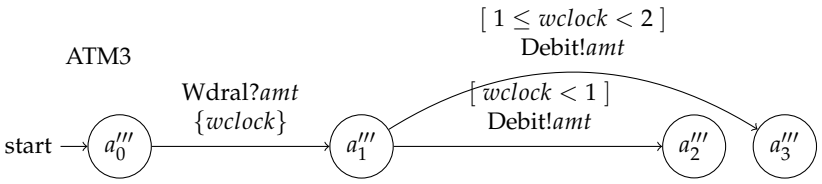
$$(\exists_{IFresh(ec_1)}.\pi(ec) \wedge OCond(ec)) \wedge (\exists_{IFresh(ec_1)}.\pi(ec') \wedge OCond(ec'))$$

is unsatisfiable

Example 2.3.2 (Deterministic TIOSTS)**.** In each following example of TIOSTS, amt and fee are natural numbers.



[5] Given an ec, we suppose that we always use the same fresh variable $OFresh(ec)_c = \chi$ to symbolically represent the observable values from ec emitted or received on channel c. Similarly, as we use the same duration for all successors of ec, we will systematically associate a duration denoted by $OdFresh(ec) = z$, which represents therefore the delay of occurrence of its successors.



Deterministic verification of ATM1. The symbolic execution of ATM1 is given in Figure 2.14. Let us consider the following data:

$IFresh(ec_1) = \{amt_0, fee_0\}$,
$\pi(ec_3) = true$,
$OCond(ec_3) = (\chi_1 = amt_1)$,
$\pi(ec_4) = (fee_0 > 0)$,
$OCond(ec_4) = (\chi_1 = amt_1 + fee_0)$.

According to the previous definition, ATM1 is deterministic if and only if the following is unsatisfiable: $(\exists_{amt_0,fee_0}.\ \chi_1 = amt_1) \wedge (\exists_{amt_0,fee_0}.\ fee_0 > 0 \wedge \chi_1 = amt_1 + fee_0)$ where χ_1 is a new fresh variable representing the value emitted on channel Debit. This formula is not satisfiable because of $amt_1 < amt_1 + fee_0$ (since $fee_0 > 0$). We can deduce then that ATM1 is a deterministic TIOSTS.

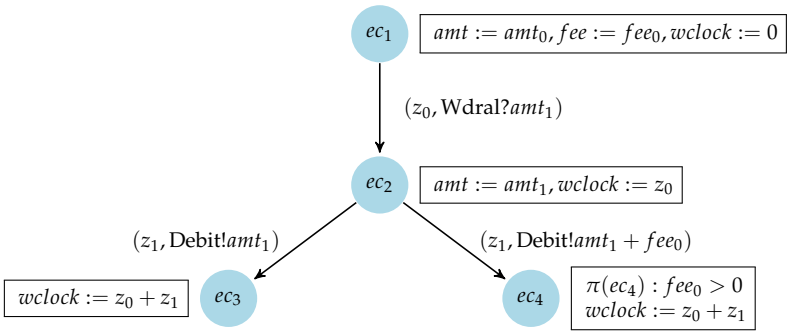


FIGURE 2.14: Symbolic execution of ATM1

Deterministic verification of ATM2. The symbolic execution of ATM2 is given in Figure 2.15. In a similar way, ATM2 is deterministic if and only if the following is unsatisfiable: $(\exists amt_0, fee_0.\ \chi_1 = amt_1) \wedge (\exists amt_0, fee_0.\ \chi_1 = amt_1 + fee_0)$ where χ_1 is a new fresh variable representing the value emitted on the channel Debit. This formula is satisfiable for $fee_0 = 0$. So, ATM2 is a non-deterministic TIOSTS.

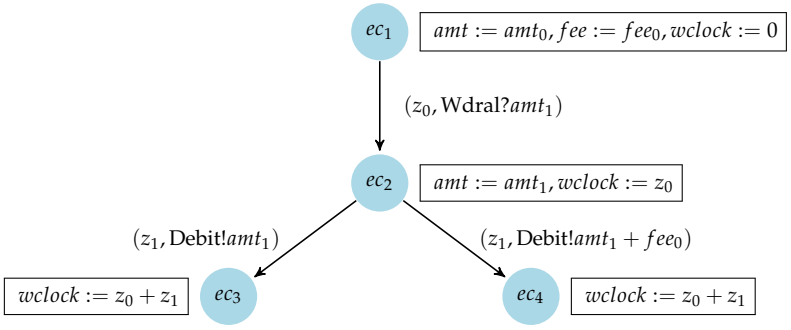


FIGURE 2.15: Symbolic execution of ATM2

Deterministic verification of ATM3. Let us verify whether or not the TIOSTS ATM3 is deterministic. The following formula which is deducted from the definition is considered: $(\exists amt_0, fee_0.\ \chi_1 = amt_1 \wedge 1 \leq z_1 < 2) \wedge (\exists amt_0, fee_0.\ \chi_1 = amt_1 \wedge z_1 < 1)$ where χ_1 is a new fresh variable representing the value emitted on the channel Debit. This formula is unsatisfiable since we cannot have a value of z_1 satisfying the formula. Indeed, the first part of the formula is satisfiable if $1 \leq z_1 < 2$, whereas the second part of the formula is satisfiable if $z_1 < 1$, which implies an empty set of values for z_1. Therefore, the TIOSTS ATM3 is non-deterministic.

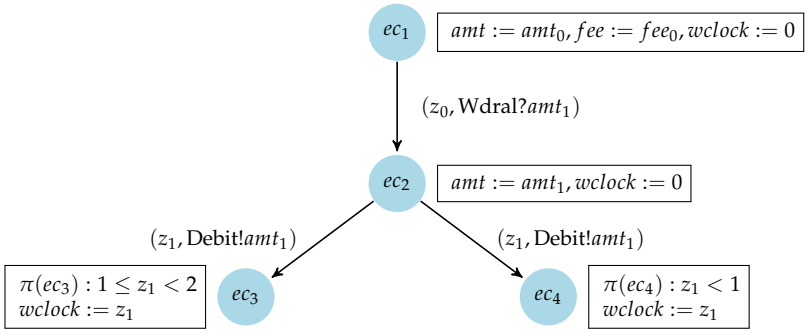


FIGURE 2.16: Symbolic execution of ATM3

Deterministic verification of the TIOSTS ATM. Now, we consider the symbolic execution tree of TIOSTS ATM in Figure 2.13, we state that for a given symbolic path $p = ec_1 \dots ec_n$, we cannot have two new execution contexts ec_r and ec_s such that $pec(ec_r) = pec(ec_s)$ and they satisfy the formula in the definition of a deterministic TIOSTS. We illustrate this formula through an example, let suppose that $p = ec_1 \cdot ec_2 \cdot ec_3 \cdot ec_4 \cdot ec_5$, from ec_5, the extension of p with ec_6 and ec_7 gives us the following symbolic timed traces:

$(z_0, \text{Init?}) \cdot (z_1, \text{Wdral?}amt_1) \cdot (z_2, \text{Debit!}(1, amt_1 + fee_0, 1)) \cdot (z_3, \text{Auth?}(rid\_ret_2, stat_2, mid\_ret_2)) \cdot (z_4, \text{Cash!}amt_1)$ for $p.ec_6$

$(z_0, \text{Init?}) \cdot (z_1, \text{Wdral?}amt_1) \cdot (z_2, \text{Debit!}(1, amt_1 + fee_0, 1)) \cdot (z_3, \text{Auth?}(rid\_ret_2, stat_2, mid\_ret_2)) \cdot (z_4, \text{Abort!})$ for $p.ec_7$

which cannot satisfy the formula in the definition of a deterministic TIOSTS because the symbolic actions in ec_7 and ec_8 contain two different communication channels. In a general way, we note that from any state in TIOSTS of ATM, we cannot have two transitions which share a same communication channel, therefore for a given symbolic path $p = ec_1 \dots ec_n$ and two distinct execution contexts ec_i and ec_j which are results of symbolic execution of two distinct transitions which could be fired from ec_n, we always have $act(ec_i) \neq act(ec_j)$. Consequently, two distinct symbolic paths cannot give a same symbolic timed trace and obviously a same timed trace. This property is verified for all symbolic paths in symbolic execution of ATM, therefore this TIOSTS is deterministic.

For the same reasoning which has been applied to the TIOSTS of ATM, we can state that the TIOSTS of Bank is also deterministic.

2.3.4 Enrichment by quiescence

A system replies to its environment by sending outputs. However, it cannot always emit an output from any given state [71, 33, 70, 5]. In this subsection, we present the quiescence enrichment in order to describe this situation when either there is a lack of transitions labeled by an output from the system or the system cannot reply due to the dissatisfaction of constraints. We remark that our quiescence enrichment is applied only to deterministic TIOSTS. To represent quiescence enrichment, we introduce the symbol _ to denote the absence of reaction observed by the tester.

Definition 2.3.2 (Quiescence enrichment)**.** Let us denote for any ec of $\mathbb{G}$, $React(ec) = \{ec' \mid pec(ec') = ec \wedge act(ec') \in O(\Sigma_F)\}$. The quiescence enrichment of $SE(\mathbb{G}) = (ec_1, \mathbb{EC})$ is denoted by $SE(\mathbb{G})_\delta = (ec_1, \mathbb{EC}_\delta)$ with $\mathbb{EC} \subseteq \mathbb{EC}_\delta$. The set $\mathbb{EC}_\delta$ is enriched by new execution contexts as follows, for any execution context $ec \in \mathbb{EC}$ we define a new one ec_δ such that: if we denote $z = OdFresh(ec)$, we let $\pi_\delta(ec) = true$ if $React(ec) = \emptyset$, and otherwise $\pi_\delta(ec) = \bigwedge_{ec' \in React(ec)}(\forall_{IFresh(ec_1)}.(\forall_z.\neg(\pi(ec') \wedge OCond(ec))))$,

- $q(ec_\delta) = q(ec)$,
- $\pi(ec_\delta) = \pi_\delta(ec)$,
- $\lambda(ec_\delta) = \lambda(ec)$,
- $ev(ec_\delta) = (z, \_)$,
- $pec(ec_\delta) = ec$.

The introduced contexts ec_δ are reachable from a given ec only if no transition labeled by an output can be executed anymore due to unsatisfiable data and/or time constraints. The path condition $\pi_\delta(ec)$ states that for all execution contexts ec' obtained from ec by executing a transition labeled by an output, whatever the delay is, the path condition $\pi(ec')$ to reach ec' cannot be satisfied.

Example 2.3.3 (Quiescence enrichment)**.** In this example, let us enrich the quiescence for the TIOSTS ATM. For this purpose, we consider the symbolic tree in Figure 2.13.

- From ec_1, the set $React(ec_1)$ is defined as $\emptyset$, the system is quiescent as it always waits for an input from environment. Hence, we enrich systematically the quiescence from the execution context ec_1. The elements of $ec_{\delta 1}$ as $e(ec_{\delta 1}), \pi(ec_{\delta 1}), \lambda(ec_{\delta 1})$ are the same as ec_1, except the symbolic event $ev(ec_{\delta 1}) = (z_1, \_)$ and the predecessor $pec(ec_{\delta 1}) = ec_1$.

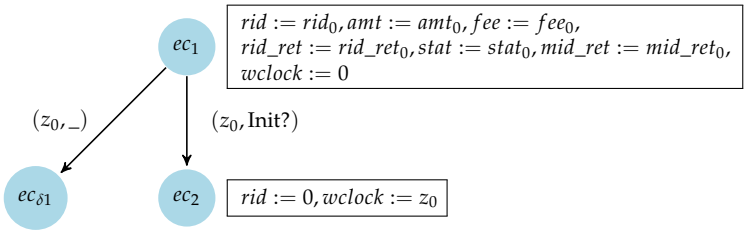


FIGURE 2.17: Enrichment by quiescence from ec_1

- From ec_3, the set $React(ec_3)$ is defined as $\{ec_4\}$. Let us consider the following data:
  $OdFresh(ec_3) = z_2$,
  $IFresh(ec_1) = \{rid_0, amt_0, fee_0, rid\_ret_0, stat_0, mid\_ret_0\}$,
  $\pi(ec_4) = (z_2 \leq 1)$,
  $OCond(ec_3) = true$.
  The path condition of quiescence enrichment from ec_3 (if exists) is defined as: $\pi(ec_{\delta 4}) = \forall_{IFresh(ec_1)}.(\forall_{z_2}.\neg(z_2 \leq 1))$. This condition is never satisfiable if z_2 takes a value which is strictly less than 1. In other words, the system is supposed to always provide a reply on channel Debit within the specified duration (e.g. 1). The quiescence enrichment cannot be applied from the context ec_3.

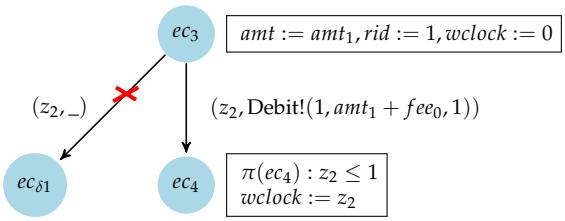


FIGURE 2.18: Enrichment by quiescence cannot be applied from ec_3

2.4 Implementation in Diversity

Diversity is an extensible tool for the development of model-based formal analyses using symbolic execution [3]. The tool has become open source since 2016 and is available for download at the following url `https://projects.eclipse.org/proposals/eclipse-formal-modeling-project`. It comes with an expressive entry language that captures a wide range of classical models semantics. One major application domain for Diversity is Model-Based Testing (MBT), the development of the tool has been mainly driven by needs arising from MBT, such as time modeling, customizable tools or selection criteria. The tool has an Eclipse-based GUI which provides a textual editor for its entry language, and exposes some built-in analysis modules such as: exhaustive exploration with stopping criteria [33] (all-paths of length k, inclusion criterion, ...), test offline oracle [5], behavior selection heuristics [3] ...

Besides the existing modules, the customizability of Diversity allows advanced users to create and define a new analysis by designing and implementing their own modules. Whatever their purpose, the implementation of a new analysis module is possible thanks to an extensible architecture of the symbolic execution engine of Diversity *symbex* by a visitor-like pattern. The symbex engine is developed in C++ language so-as the analysis modules. One (or more) module(s) can be called when symbex is executed on the (textual) model, in order to apply their analyses during the symbolic exploration.

We have implemented a first module which allows the computation of a symbolic path which covers consecutive transitions. Those are meant to represent a *test purpose*, which is a particular behavior that we would like to test in a given subsystem (this notion will be discussed more clearly in next chapter). In spite of being a very simple module and there are already many other coverage modules such as transition coverage, behavior selection, etc., our module allows to verify the feasibility of a sequence of consecutive transitions, and is used to select a test purpose during the process of test case generation that is presented in Chapter 3.

In the first part of the section, we overview the encoding of the ATM-Bank system using the tool entry language. In the second part, we present the extensible symbex engine of the tool illustrated by our consecutive transitions coverage module. And finally, in the third part of this section we discuss the application of this coverage module on the subsystem ATM of the distributed system ATM-Bank.

2.4.1 Textual models of the ATM-Bank system

Diversity provides a pivot textual language called xLIA (eXecutable Language for Interaction and Assembly) introducing a set of communication and execution primitives allowing one to encode a wide class of dynamic model semantics, (timed) Symbolic Transition Systems (STS) [29, 33], (communicating) Finite State Machine (FSM) [19, 30], data flow and Synchronous Data Flow (SDF) [50], SDL (ITU-T) [40], UML/SysML (OMG) [63], ... An xLIA system model can be atomic, a state machine, or compositional or hierarchical. We show next how we encode the ATM-Bank system using xLIA.

The distributed system ATM-Bank is composed of many subsystems, each of them could be ATM or Bank. In the following, we show how to declare different elements for the TIOSTS ATM such as variable, clock,

communication channel, transition. Figure 2.19 presents the declaration of the communication channels and variables. All channels are declared with the modifier `public` by using keyword `port` whereas all variables are declared in private (e.g. modifier `private`) using keyword `var`. Similar to variables, each channel is well-typed, e.g. Wdral only accepts the value of type `pos_integer` (i.e. positive integer).

```
lifeline model timed statemachine < and > ATM {

    @public:
        port input Init;
        port input Wdral( pos_integer );
        port input Auth( pos_integer, Response, pos_integer );
        port output Debit( pos_integer, pos_integer, pos_integer );
        port output Log( pos_integer, Response, pos_integer );
        port output Abort;
        port output Cash( pos_integer );

    @private:
        var pos_integer rid;
        var pos_integer amt;
        var pos_integer fee;
        var Response stat;
        var pos_integer rid_ret;
        var pos_integer mid_ret;
        var pos_integer ATM_ID = 1;
        var clock urational wclock;
```

FIGURE 2.19: Channel and variable declaration of ATM

In Figure 2.20, we present transition on channel Debit of ATM and Bank. The subsystem ATM sends a request on the channel Debit containing the request identifier *rid*, an amount including the requested one *amt* and a charge fee, and an ATM machine identifier ATM_ID. This request is constrained by the guard over *wclock*, meaning that it is fired when the value of this clock does not exceed 1 time unit. To receive this kind of request, the subsystem Bank stores the values that have been received in variables tid, x, aid. This transition is fired if the value of *dclock* satisfies $dclock + 6 < 86400$ (i.e. the value of `TWENTY_FOUR_HOUR` is 86400 time units). We remark that Debit is declared as an output channel for ATM and as an input channel for Bank.

```
transition tr_3a --> a3 {
    output Debit( rid, amt + fee, ATM_ID );
    guard( wclock <= 1 );
}
```

```
transition tr_2b --> b2 {
    input Debit( tid, x, aid );
    guard( dclock + 6 < TWENTY_FOUR_HOUR );
    pclock := 0;
}
```

FIGURE 2.20: Transition on Debit of ATM (left) and Bank (right)

The Diversity tool proposes also a mechanism to create many instances of a state machine through keyword `instance`. Diversity's users can also modify value of a variable dedicated to a proper instance declaration. For example, Figure 2.21 shows the declaration of two TIOSTS ATM1 and ATM2 based on the basic template ATM. The machine identifier for ATM1 and ATM2 are respectively declared as 1 and 2.

```
instance< ATM > ATM1
{
    ATM_ID = 1;
}

instance< ATM > ATM2
{
    ATM_ID = 2;
}
```

FIGURE 2.21: Declaration of two instances ATM1 and ATM2

Depending on the system structure (atomic, compositional or distributed system), Diversity offers many ways in order to connect communication channels in xLIA. In the figure below, let us introduce three kinds of connection from left to right: 1) when the subsystem ATM1 is executed in isolation, the channel Debit is connected to the environment, denoted as `env` 2) when the system is compositional, an output on channel Debit from ATM1 is immediately consumed by Bank (synchronous communication). The keyword `rdv` is used to declare a synchronous communication 3) when the system is distributed, then an output on channel Debit from ATM1 is stored in a waiting queue of type `fifo` for being consumed lately (asynchronous communication). The keyword `fifo` is used to describe the nature of a waiting queue (other natures `lifo`) and `<*>` is used to designate its unbounded capacity of storing received messages.

```
@com:
    connect< env >{
        output ATM1->Debit;
    }
```

```
@com:
    connect< rdv >{
        output ATM1->Debit;
        input Bank->Debit;
    }
```

```
@com:
    connect< fifo <*> >{
        output ATM1->Debit;
        input Bank->Debit;
    }
```

FIGURE 2.22: Different kinds of connection in Diversity

In Figure 2.23, the right subfigure shows the declaration of a multicast connection on channel Auth between Bank and two instances ATM1 and ATM2. Bank sends a message in multicast to both ATM1 and ATM2 which store the message received in their local waiting queue `bufferATM`. This waiting queue must be previously declared inside of the state machine ATM as in the left subfigure. Similarly, `<*>` indicates that the buffer has an unbounded capacity of storing received messages.

```
@public:
    buffer fifo<*> bufferATM;
```

```
@com:
    connect< multicast >{
        output Bank->Auth;
        input < buffer< ATM1.bufferATM > > ATM1->Auth;
        input < buffer< ATM2.bufferATM > > ATM2->Auth;
    }
```

FIGURE 2.23: Multicast connection in Diversity

For a given distributed system, the mechanism of scheduling represents a computational model that allows to execute behaviors of subsystems running in parallel. We have many kinds of scheduling between subsystems, such as sequence, interleaving, etc. In Diversity, in order to define a scheduling, we use the keyword `schedule` to declare a scheduling section. In case of the distributed system ATM-Bank, we choose to define its scheduling by using interleaving operator (`|i|`) to illustrate possible interactions between subsystems. In Figure 2.24, ATM1, ATM2 and Bank are executed by considering every possibility: transitions of three subsystems are executed in any possible order when operational.

```
@schedule{|i|
    run ATM1;
    run ATM2;
    run Bank;
}
```

FIGURE 2.24: Interleaving scheduling in Diversity

2.4.2 The tool customizable symbex engine

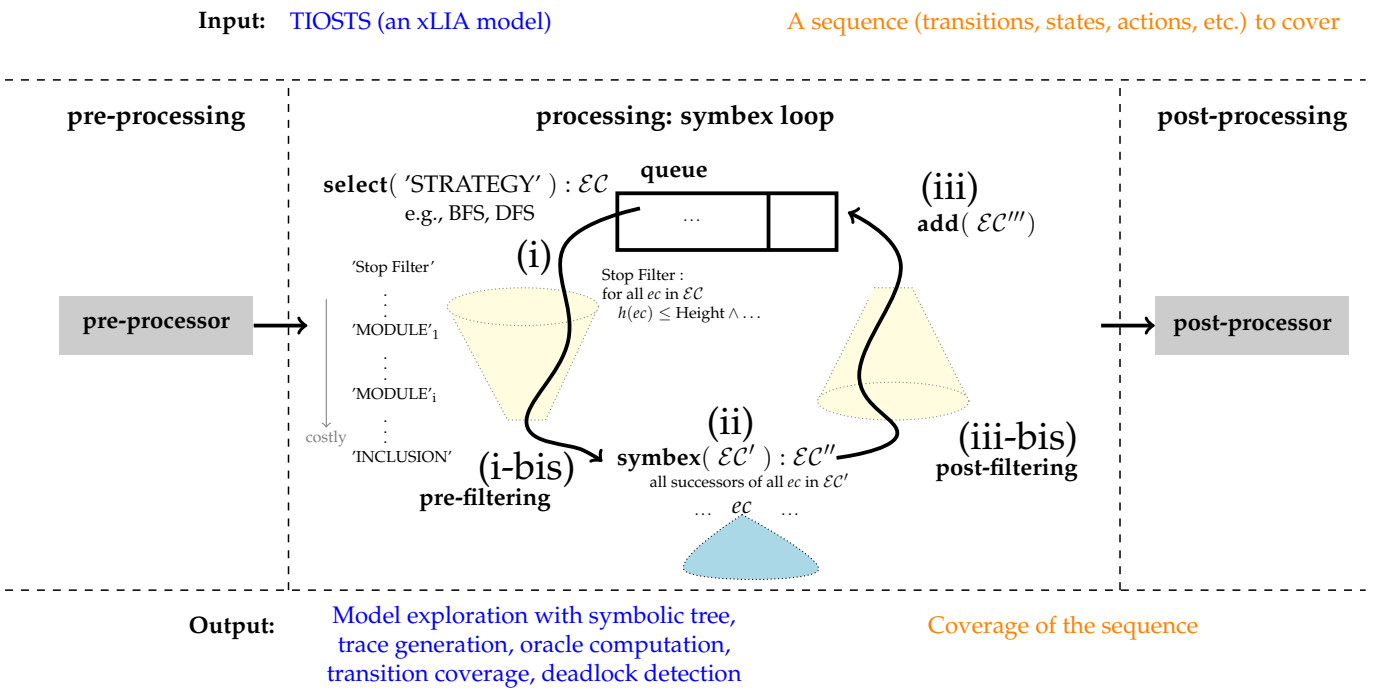


FIGURE 2.25: Overview of Diversity's symbex engine

The Diversity tool is an multipurpose platform allowing different analyses for models using symbolic execution. From a TIOSTS model, one can perform many analyses such as model exploration, trace generation or verdict computation, etc. Diversity has been designed in a way which facilitates the implementation of a new analysis module. Advanced users may develop their own modules for proper objectives. In Figure 2.25, we illustrate the overview of the symbex engine in Diversity based on which each module is implemented.

Firstly, we present a priori step that could be arisen at the beginning of the symbex engine, called **pre-processing**. The pre-processing step occurs before the symbolic execution loop. Its main purpose is to construct initial context (i.e. ec_1) and enqueue this context. It may also transform the model for the formal treatments to be applied (for example adding sink states in the model).

The symbex engine consists in executing symbolically the system model. During execution, it maintains a waiting queue containing EC for the next execution step. At the beginning, the process initializes the queue with the initial execution context (i.e. ec_1). The symbolic execution loop is composed of five main following steps:

- **(i) Selection of EC**: According to a strategy chosen by the user (BFS: Breadth-First Search, DFS: Depth-First Search, RFS: Random-First Search or ALL: All contexts are selected), the symbex selects from the waiting queue one or more ECs for Pre-Filtering step. The set of selected ECs is denoted as $\mathcal{EC}$.
- **(i-bis) Pre-Filtering** - Analyses for filtering EC for execution step: This step consists in performing some analyses to reason on selected ECs before computing their successors. An EC that does not pass these analyses will be added to the symbolic tree. The contexts that pass analyses are stored in the set $\mathcal{EC}'$.
- **(ii) Symbex** - Execution step for computing EC-successors: We compute successors of any EC belonging to $\mathcal{EC}'$ in Pre-Filtering step. The set of successors is denoted as $\mathcal{EC}''$.

- **(iii-bis) Post-Filtering** - Analyses for filtering EC-successors for queuing for next execution step: The selected ECs from Pre-Filtering step are added to symbolic tree. This step is used to perform some analyses on their successors obtained from symbex step. An EC-successor that passes these analyses will be added to the set $\mathcal{EC}'''$. Otherwise, it will be inserted in the symbolic tree.

- **(iii) Addition of EC-successors**: All the successors resulting from the Post-Filtering step (i.e. the set $\mathcal{EC}'''$) are added to the waiting queue and the symbolic execution engine starts over from step (i)

The module 'INCLUSION' stops the symbolic execution of any ec (at pre-filtering) when it is included semantically in another already computed one ec' in the symbolic tree [33]: this means that any transition which can be fired from ec, can also be fired from ec'. Calling this module is costly as inclusion checking has a high computational cost in comparison with for instance 'Stop filter' which checks if the height of an ec (distance to ec_1 the root of the symbolic tree) reaches maximum height bound ($h(ec) \leq Height$).

After the symbolic execution loop (the symbolic execution tree is computed at this step), symbex engine can process a posteriori step called **post-processing**, which consists in restoring the model in its original form whenever it was modified in the **pre-processing** step, and performing some processing in the computed symbolic tree. Two steps **pre-processing** and **post-processing** occurs respectively before and after the symbolic execution loop, therefore they are not part of the symbolic execution loop.

2.4.3 Module of test purpose selection

In this subsection, we introduce the module of coverage called TESTPURPOSE_SEL that has been implemented in our work to cover a sequence of consecutive transitions in a TIOSTS. The objective of this module is to verify the feasibility of this sequence of consecutive transitions, i.e. there exists at least a timed trace satisfying the formulas constraining the transitions in the sequence. Because of this functionality, TESTPURPOSE_SEL will be applied to a sequence of consecutive transitions before applying the test case generation process (this will be discussed in next chapter). In our module, the selection strategy 'ALL' is required, meaning all contexts from the waiting queue are selected.

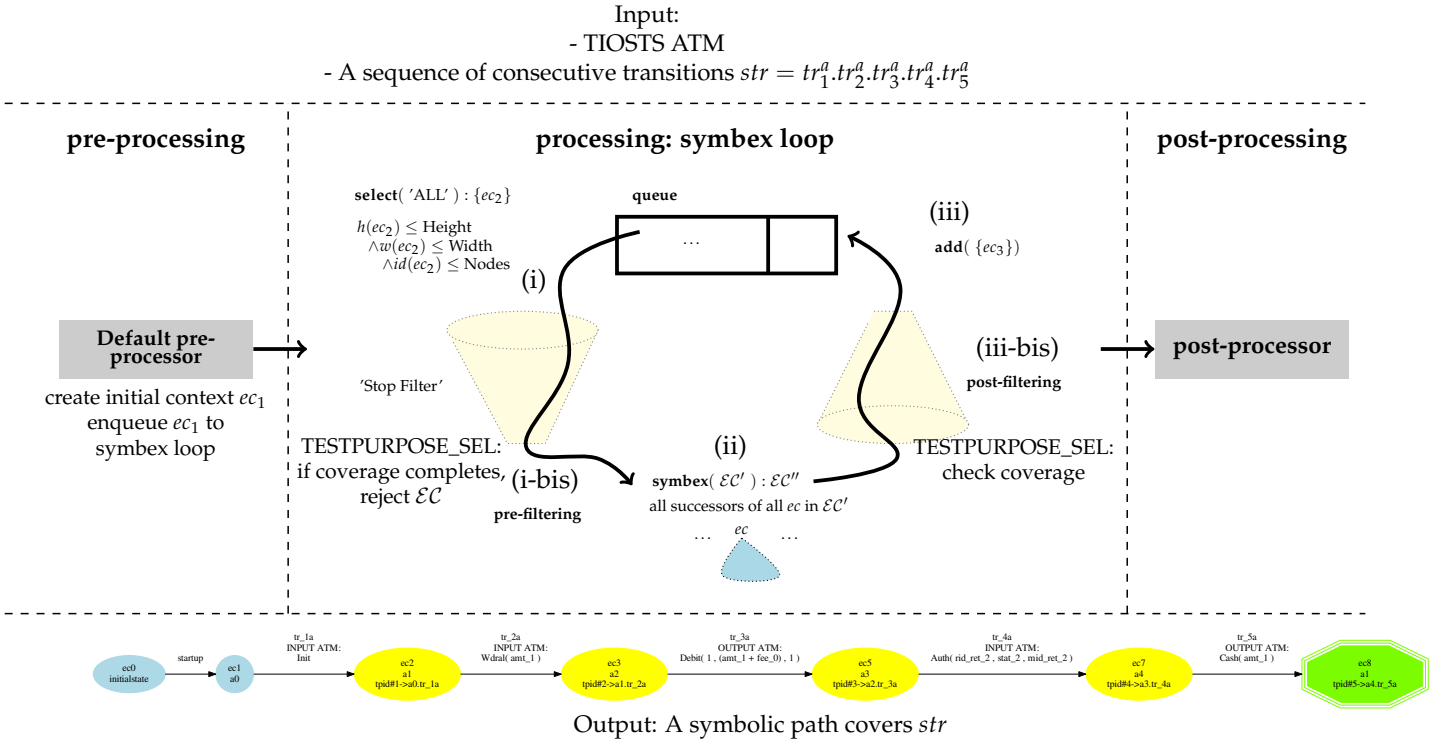


FIGURE 2.26: Module of TESTPURPOSE_SEL in symbex

Let suppose that we need to cover the following sequence $str = tr_1^a \dots tr_5^a$ (Figure 2.27) which describes the following scenario:

1. At the beginning, the ATM machine receives an initialization message;
2. Lately, a client asks for an amount at the ATM;
3. The request is then transferred on the channel Debit;
4. Subsequently, the ATM machine receives the withdrawal authorization;
5. Finally, it returns cash to the client.

```
TESTPURPOSE_SEL 'check feasibility of a test purpose' {
    Sequence [
        transition = "tr_1a"
        transition = "tr_2a"
        transition = "tr_3a"
        transition = "tr_4a"
        transition = "tr_5a"
    ]
    queue 'defining the exploration/search strategy' [
        strategy = 'ALL'
    ] // end queue
}
```

FIGURE 2.27: Screenshot from a workflow file of TESTPURPOSE_SEL module for ATM

The application of the module TESTPURPOSE_SEL to *str* produces a symbolic path as illustrated in Figure 2.26. The following graphical conventions are used to denote different kinds of contexts: the initial execution context is colored in blue, a context permitting to cover the transitions in the sequence *str* is colored in yellow, except the one that covers the last transition is colored in green.

For more information about the implementation of this module, please refer to the following website: https://projects.eclipse.org/proposals/eclipse-formal-modeling-project

2.5 Conclusion

In summary, this chapter introduced a formal model (TIOSTS) used to describe expected behaviors of reactive systems. Then, we introduced symbolic execution techniques applied to the TIOSTS to derive symbolic behaviors. Lately, we presented a new kind of coverage called TESTPURPOSE_SEL, which is developed in the Diversity tool, in order to verify the feasibility of a sequence of consecutive transitions. The usefulness of this module will be illustrated in our process of test case generation in subsequent chapters. Of course, there are many other coverage strategies which have been already implemented in Diversity such as coverage of all transitions or behavior selection. However, we decided to implement this coverage since it facilitates the selection of a test purpose in our approach.

Chapter 3

Test cases derivation in the distributed context

Contents

In this chapter, we present our contribution for testing unitary (sub)systems while being running in a distributed context: in practice some of their stimulations or inputs will not be provided by the co-localized testers, and will be provided by other distant subsystems. The contribution consists in: i) the definition of the generation of online test cases from the symbolic models and ii) their implementation in the symbolic analysis tool Diversity. In order to facilitate the reading, let us introduce the organisation of this chapter:

- Section 3.1 recalls Model-Based Testing principle for reactive systems. This section also illustrates two principal categories of conformance testing: the offline approach and the online approach;
- Then, Section 3.2 introduces the mathematical conformance relation tioco which states the meaning that we give to the verdicts (Pass, Fail, ...) in the test case.
- Section 3.3 gives the rules for the generation of online test cases, these are given in the form of deterministic (acyclic) TIOSTS. The generation is driven by a particular behavior of the system that we want to test, the so-called *test purpose*. We illustrate the generation rules on the ATM example in Section 3.4;
- Section 3.5 is devoted to implementation: We first present the module that has been developed to implement the rules. This module has been implemented using the extension mechanisms of the symbolic

process in Diversity; and then, we give few elements on the validation of the generated test cases with respect to the conformance relation tioco by exploring the test case behaviors (being a TIOSTS itself) using diversity and in particular analysing verdicts reachability;

- We review some related model-based approaches for testing reactive systems in Section 3.6. And finally, we conclude the chapter in Section 3.7.

3.1 Model-Based Testing of reactive systems

Model-Based Testing (abbreviated as MBT) is a testing technique which uses models to automate the test cases generation or definition and eventually their executions. When it comes to testing reactive systems, many MBT approaches use models as a reference to test their intended behaviors. Figure 3.1 describes the generic process of MBT for reactive systems.

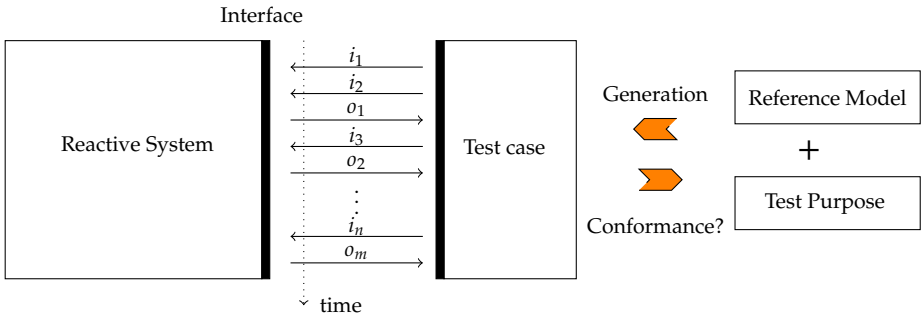


FIGURE 3.1: Model-Based Testing Process

MBT consists of the following activities:

- Reference models : A model M represents intended behaviors of a reactive system. There are different kinds of behavioral models for such system: a model can be a Labelled Transitions System (LTS) [77, 78], a Symbolic Transitions System (STS) [33, 5] or a Finite State Machine (FSM) [30, 84, 59], ... (see Chapter 2). Such a model is constructed often based on high level requirements of the system. In black box testing, the model expresses the interaction of the system with its environment, in terms of sequences of inputs, outputs, and, eventually observed delays in-between, while abstracting as much as possible its internal implementation details.

- System Under Test (abbreviated as SUT) : It refers to a system that is going to be tested for correct behaviors. Those are characterized by a model in the MBT approach. In Figure 3.1, the reactive system is an SUT.

- Test purpose : A test purpose TP represents a particular behavior of the model M that the test case would like to observe on a SUT in order to verify whether or not this SUT behaves correctly.

- Test case generation: This activity consists in generating test cases from the model M, which is driven by the test purpose TP, the latter may be either defined as sub-behavior of the model, for instance selected from the model using coverage criteria [22] or have been constructed in the previous activities, for instance a specific behavior (or property) defined independently of the model [1]. The test cases are constructed in such a way that it can interact with the SUT. Indeed, a generated test case can be plugged on the SUT in order to send the stimulations to the SUT and observe its outputs, i.e. inputs

are computed to stimulate the SUT and outputs are compared to the ones of the model. There are two categories of test case generation: offline [5, 32] and online [1, 33] that we will talk in the subsequent sections. Roughly speaking, offline approach consists in generating in advance the sequences of inputs and delays and each sequence is considered as a test case, whereas in online approach, next inputs (and their adequate submission delays) are computed on-the-fly based on previous interactions and with the objective to remain as much as possible within the test purpose, eventually by calling a solver, a verdict is issued at the earliest to stop the test execution.

- Test execution: MBT consists in executing the test cases as the SUT is being executed with the objective to make both interact. Usually test cases are therefore transformed into an executable form, for instance as TTCN scripts [85] executable in the test execution tool Titan [75].

- Conformance check: This activity is about checking that the behaviors of an SUT conforms to the reference model. In fact, by executing a test case, outputs are provided by the SUT in response to test case stimulations, we have then to analyze the correctness of these outputs and their occurrence delays. Such correctness is often stated by a mathematical conformance relation, a well-known one is the *input output conformance relation* (abbreviated as ioco) [78] and its timed extension (tioco) [46, 5]. This relation has become a standard in system verification and has allowed the development of many test case generation and verdict computation from models.

In the context of MBT, most of testing approaches could be categorized into offline and online testing that we discuss in the following section.

3.1.1 Offline approach

In the offline approach [5, 10, 32], a test case is derived from the combination of the test purpose and the model. From a test purpose, the offline approach pre-computes a sequence composed of inputs (denoted by $i_1, \ldots, i_n$) and their submission delays. During the test execution, the corresponding sequence of outputs (denoted by $o_1, \ldots, o_m$) is recorded. This sequence is then merged with the sequence of inputs based on temporal information as being occurrence delays of inputs or outputs (or timestamps) to reconstruct the execution sequence. The last step consists in analyzing the obtained sequence in order to compute the test verdict w.r.t the model. This steps is performed after the test execution, thus the approach is called offline testing. The generic process of offline testing is shown in Figure 3.2.

We remark that in this approach, all inputs and delays are computed in advance. The advantage of offline approach is that it facilitates the test execution because all time guards and data guards have been completely resolved during test case generation process, i.e. the sequence of inputs is completely pre-computed, therefore we do not need to compute new inputs to stimulate SUT. Moreover, a test case could be applied as many times as we want to test a particular behavior of the system.

There are several main disadvantages concerning offline approach. The first one is that the test case must be completely executed on the SUT to be able to compute the test verdict even though the test case is a very long sequence of inputs and delays. The verdict computation is based on a conformance relation allowing to reason system correctness. We discuss more in detail a variant of this relation in Section 3.2. The second disadvantage is the non-deterministic behavior of the SUT. It is the case when an output emitted by the SUT is not specified in the test purpose but is specified by the model. Consequently, it is not possible to conclude the test verdict as *Pass* or *Fail*. Another test verdict *Inc* is emitted in this situation, i.e. inconclusive.

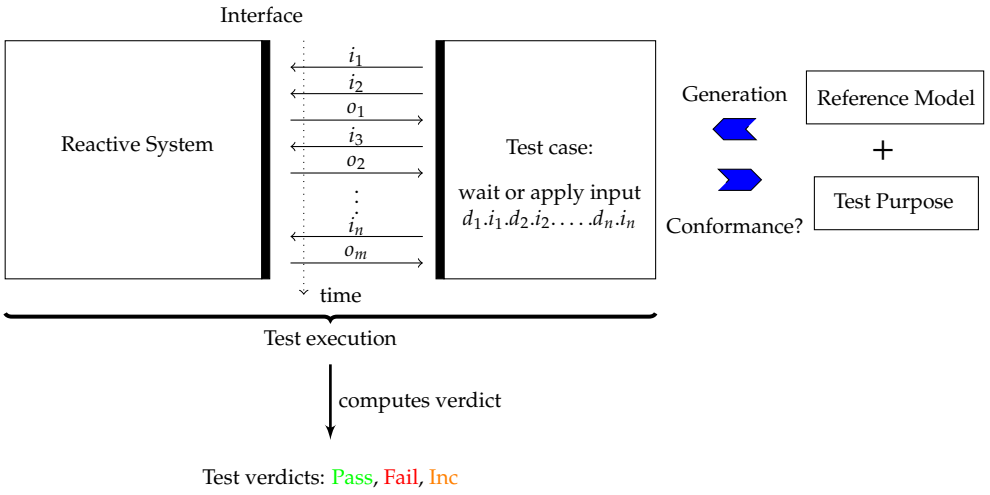


FIGURE 3.2: Offline approach

3.1.2 Online approach

The online testing combines test generation and execution [36, 37, 33]. It can also take benefit of a model and a test purpose. The online testing process is depicted in Figure 3.3. Unlike offline approach, the particularity of online approach is that inputs (their submission delays) are computed on-the-fly (i.e. during the test execution). The computation of the next stimulation is based on the previous observations. Indeed, the input is generated on-the-fly and then it is sent to stimulate the SUT. Typically, the inputs and delays are chosen such that the interaction between the test case and the SUT is specified in the test purpose. The second particularity of online approach is that the system correctness is performed after each reception of an output produced by the SUT. Indeed, the output produced by the SUT as well as its moment of occurrence are verified directly against the model after the reception: if the behavior is specified in the test purpose, the test execution is continued (until a verdict *Pass* is emitted), if it is specified in the model but not compatible with the test purpose, a verdict *Inc* is emitted and the test execution stops, otherwise (i.e. the behavior is not expected by the model), a verdict *Fail* is emitted and the test execution stops also.

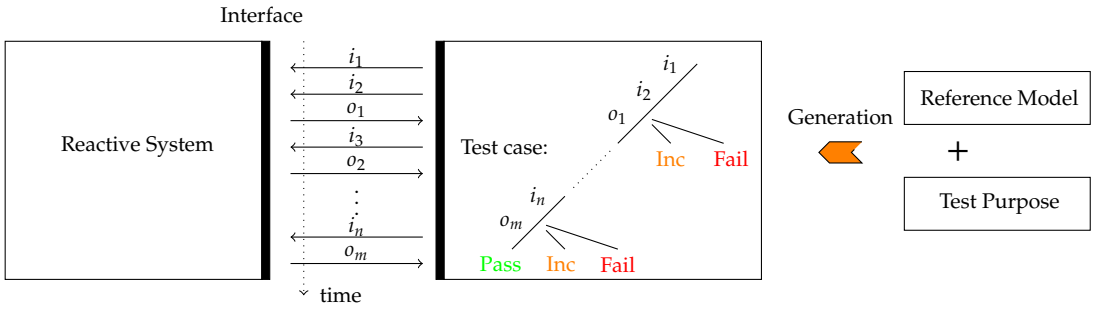


FIGURE 3.3: Online approach

An advantage of the online approach compared with the offline one is that it can target very long behaviors of an SUT without deviating from the test purpose as the test case adapts the next input computation with respect to previous observations, and in favor for targeted future outputs to be observed in the test purpose. Besides, the test case does not need to be completely executed on the SUT to reveal a non-compliant

behavior of SUT, i.e. it can help us detect early failures and proceed by executing other test cases in the test campaign.

3.2 Timed conformance relation

In this section, firstly, we characterize a Localized System Under Test (abbreviated simply as LUT). Secondly, we present tioco, which is a conformance relation relating the correctness of a given timed trace resulting from the execution of an LUT against its reference model TIOSTS, whose semantics is given by timed traces.

3.2.1 Localized System Under Test

As we deal with Localized Systems Under Test in the distributed context, we need to distinguish between inputs locally received from some users (or test cases) and inputs received from other distant subsystems via the network. Therefore, we consider the following partition of C^{in}, the set of input channels:

$$C^{in} = CC^{in} \amalg UC^{in}$$

where CC^{in} is the set of controllable input channels and UC^{in} is the set of uncontrollable input channels. A channel in CC^{in} is used to receive messages from the test case (hence controllable for the test case), whereas a channel in UC^{in} is used to receive messages emitted by other subsystems (uncontrollable for the test case).

It might happen that the test case does not wait long enough to observe a reaction of a Localized System Under Test because the delay is not known in advance (and in whole generality not bounded). We recall that to represent this situation, we introduced in Section 2.3.4 the symbol _ to denote the absence of reaction observed by the tester. An event $(d, \_)$ will be used to denote that after waiting delay d, no reaction of the Localized System Under Test is observed. $Evt(\{\_\})$ denotes the set of all such events.

Notation 3.2.1. For any $ev = (d, \_) \in Evt(\{\_\})$, $delay(ev)$ and $act(ev)$ denote d and _ respectively.

A Localized System Under Test is as a black box, and defined as a set $LUT \subseteq (Evt(C) \cup Evt(\{\_\}))^*$ satisfying the following hypotheses for any $\sigma_1, \sigma_2 \in Evt(C)^*$ and $ev \in Evt(C)$:

- stable by prefix:
$$\sigma_1.\sigma_2 \in LUT \Rightarrow \sigma_1 \in LUT$$
- quiescence: for any $d < delay(ev)$,
$$\sigma_1.ev \in LUT \Rightarrow \sigma_1.(d, \_) \in LUT$$
- input complete: for any $d < delay(ev)$, $c \in CC^{in}$, $v \in M$,
$$\sigma_1.ev \in LUT \Rightarrow \sigma_1.(d, c?v) \in LUT$$

The hypothesis on quiescence states that if the LUT is waiting for an event ev whose action is not empty, then any duration which is strictly less than the delay of ev is accepted by the LUT. The hypothesis on input completeness enables LUT to receive any input on a controllable channel, i.e. an input received from the test cases, during the delay of ev.

In the following, we present tioco, timed conformance relation between a timed trace of the LUT and timed traces that can be generated from the reference model. This relation states that the LUT is in conformance with the model, if and only if after a specified sequence of events between the LUT and the test

case, any event produced by the LUT (either a reaction or an observation of a delay) must be specified in the reference model.

3.2.2 Timed Input Output Conformance relation (tioco)

The semantics of a TIOSTS $\mathbb{G}$ can be seen as timed traces (cf Section 2.2.2). Now, in order to reflect the quiescence situations that can be observed during the test execution on LUT, we define the semantics of $\mathbb{G}$, denoted by $Sem(\mathbb{G})$, as the smallest set containing $Trace(\mathbb{G})$ and such that for any $\sigma \in Evt(C)^*$, $ev \in Evt(C)$, for any $d < delay(ev)$:

$$\sigma.ev \in Trace(\mathbb{G}) \Rightarrow \sigma.(d, \_) \in Sem(\mathbb{G})$$

We stress that, as a consequence of TIOSTS semantics, when $\mathbb{G}$ is waiting for an event ev, it accepts any duration which is strictly less than the delay of ev. In the following, elements of the semantics $Sem(\mathbb{G})$ are called *sequences of events*.

We use the following reformulation of the tioco conformance relation [46] to define correctness of a LUT with respect to a TIOSTS $\mathbb{G}$:

LUT tioco $\mathbb{G}$ if and only if, for all $\sigma \in Sem(\mathbb{G})$, for any either event $ev \in Evt(C)$ with $act(ev) \in O(C)$ or $ev \in Evt(\{\_\})$, we have:

$$\sigma.ev \in LUT \Rightarrow \sigma.ev \in Sem(\mathbb{G})$$

In an informal way, the LUT conforms to the reference model $\mathbb{G}$ if after any sequence of events σ, an output event ev is produced by the LUT, then this event must be also specified by the TIOSTS $\mathbb{G}$.

Example 3.2.1. We introduce in this example many sequence of events of the ATM4 presented in Figure 3.4 in order to illustrate the conformance relation tioco. This example is a variant of the complete example introduced in Subsection 2.2.3.1. Let us suppose that ATM_ID is a constant and predefined as 1.

Let us discuss some sequence of events of some hypothetical LUTs and their conformance:

- $\sigma_1 = \sigma_1'.(1, \text{Debit!}(1, 12, 2))$ where $\sigma_1' = (2, \text{Init?}).(2, \text{Wdral?}10)$:

  σ_1' is a sequence of events of $Sem(\text{ATM4})$, however σ_1 is not ($\sigma_1 \notin Sem(\text{ATM4})$). This is because the output event $(1, \text{Debit!}(1, 12, 2))$ is not specified in ATM4 as a possible continuation of σ_1': the third element in the tuple of information transmitted on the channel Debit must have a value of 1 since ATM_ID is predefined as 1 (see Figure 3.4). Therefore, this is an erroneous unspecified output.

- $\sigma_2 = \sigma_2'.(3, \text{Debit!}(1, 14, 1))$ where $\sigma_2' = (2, \text{Init?}).(1, \text{Wdral?}12)$:

  Once again, σ_2 is not a sequence of events in $Sem(\text{ATM4})$, since the delay between the last event of σ_2' and the next event on the channel Debit is measured as 3 time units, which exceeds the delay permitted by the transition from state a_2 to a_3 (conditioned by $wclock \leq 1$).

- $\sigma_3 = \sigma_3'.(3.4, \_)$ where $\sigma_3' = (3, \text{Init?}).(2, \text{Wdral?}20).(0.5, \text{Debit!}(1, 22, 1)).(1.6, \text{Auth?}(0, \text{ACCEPT}, 1))$:

  It is straightforward to find that σ_3' is specified by ATM4, let verify if the event $(3.4, \_)$ is allowed by the model. After σ_3', the system reaches the state a_4', and the only possible behavior is the emission of an output on channel Cash when the guard of transition is satisfied, i.e. $wclock > 4 \wedge stat = \text{ACCEPT}$. The system LUT has elapsed 3.4 time units after σ_3' without sending anything to the environment, therefore this behavior is accepted by the model. The clock $wclock$ of LUT_3 has elapsed in total 5.5 time units (i.e. sum of 0.5, 1.6 and 3.4 since it has never been reset from the second event). According the conformance relation tioco, σ_3 holds the conformance relation and belongs to $Sem(\text{ATM4})$.

- $\sigma_4 = \sigma'_4.(1.4, \text{Auth?}(0, \text{ACCEPT}, 3))$ where $\sigma'_4 = (1, \text{Init?}).(5, \text{Wdral?}15).(0.5, \text{Debit!}(1, 17, 1))$:

  The sequence σ'_4 is specified by ATM4 and after σ'_4, the system reaches the state a_3. However, the last event of σ_4 does not correspond to an event of the transition from a_3 since the value of *mid_ret* is 3 which is different from the predefined value of the constant ATM_ID (i.e. 1). An under-specified input event cannot violate the conformance, we recall that on the other hand, specified outputs or delays in the model has to be observed.

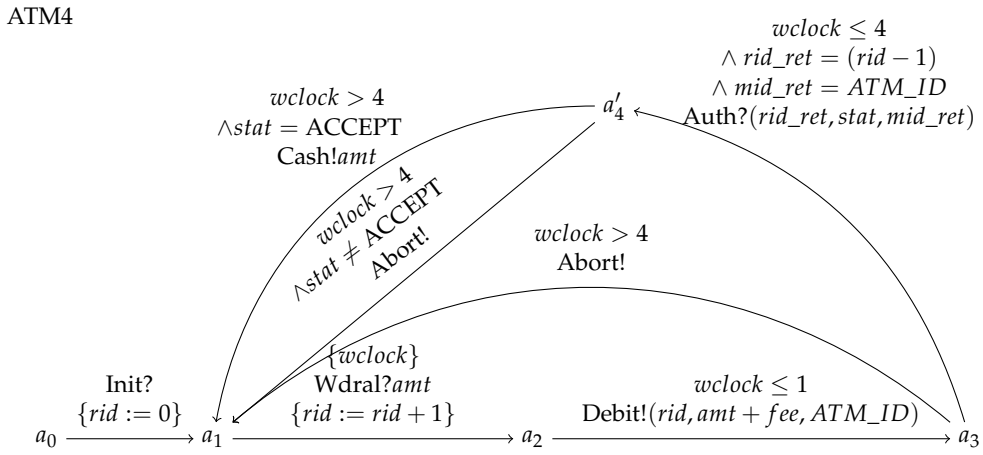


FIGURE 3.4: ATM4 ($ATM\_ID = 1$): under-specified for some internal inputs

3.3 Construction of online test case

We have presented in Section 3.1 two approaches of MBT: offline and online. Our work falls into the second category, the test case is generated in such a way it stimulates the LUT based on previous observations, and it allows the computation of verdicts on-the-fly.

In the sequel, we suppose that a TIOSTS $\mathbb{G}$ and its symbolic execution tree are previously computed. In our approach, the test case generation is based on the selection of a test purpose in the symbolic execution tree of $SE(\mathbb{G})$. The selection of a test purpose permits to characterize a particular behavior in the reference model [1, 5, 36]. In the following, we consider a test purpose of $\mathbb{G}$ as a finite symbolic path $tp \in SPaths(\mathbb{G})$.

In our work, we consider only deterministic TIOSTS for test case generation. In addition, in order to simplify the test case definition, we assume that channels are typed, which implies that the emitted or received piece of data (i.e., concrete value) via a given channel is of the same type. By default, for each execution context ec in $SE(\mathbb{G})$ such that the action of its event is an output on a channel $c \in C^{out}$, we associate ec with a fresh variable $OFresh(ec)_c$ of same type, which will be useful in test case definition to store any output value emitted by the LUT on the channel c, when the execution follows that specific execution context ec. We point out that as the test case is defined based on a given test purpose tp, variables used to store values corresponding to internal inputs are those introduced as fresh reception variables when tp is defined by symbolic execution (see symbolic execution steps in Section 2.3).

Notation 3.3.1. For any execution context ec, we denote by $Fresh(ec)$ the set of additional fresh variables that are introduced since its predecessor for the definition of ec as follows:

$$Fresh(ec) = \begin{cases} \{z, x\} & \textbf{if } ev(ec) = (z, c?x) \\ \{z\} & \textbf{if } ev(ec) = (z, c!t) \\ \emptyset & \textbf{if } ev(ec) = (0, \_) \end{cases}$$

We recall that from the remark 2.3.1, ec_1 is the considered initial context with $ev(ec_1) = (0, \_)$ and $Fresh(ec_1) = \emptyset$. For the initial context ec_1, $IFresh(ec_1)$ is the set of fresh variables used to initialize the variables, other than clocks, in ec_1 at the beginning of symbolic execution. Moreover, for a given symbolic path p, $OCond(p)$ is the identification condition of p's observable values.

For any symbolic path $p = ec_1 \dots ec_n$, let $PFresh(p) = \bigcup_{1 \leq k \leq n} Fresh(ec_k)$. Now, for any ec, we denote by $PFresh(ec)$ the set $PFresh(p)$ where p is the symbolic path that leads to ec.
The set $PFresh(ec)$ identifies the variables introduced by symbolic execution, that will be associated with concrete values when the execution of the test case against LUT reaches ec.

The notation $Var(\varphi)$ is used to denote the set of variables appearing in the definition of a formula φ.
Finally, for any decomposition of a symbolic path $p = ec_1 \dots ec_n$ into $p = ec_1 \dots ec_{k-1} \,.\, ec_k \,.\, ec_{k+1} \dots ec_n$, we denote $ec_1 \dots ec_{k-1}$ and $ec_{k+1} \dots ec_n$ respectively by $pre(ec_k)$ and $post(ec_k)$.

Let tp be a test purpose of $\mathbb{G}$. The test case generated for the test purpose tp is a TIOSTS $\mathbb{G}^{tp} = (Q^{tp}, q_0^{tp}, Tr^{tp})$ over a signature $\Sigma^{tp} = (\Omega, A^{tp}, K^{tp}, C)$ defined partially based on the signature $\Sigma = (\Omega, A, K, C)$ of the reference TIOSTS $\mathbb{G}$ as follows.
The set of clocks K^{tp} contains one clock cl used to measure the delay of each transition fired in the test case.
The set of communication channels (C) of the test case is the same as for $\mathbb{G}$: indeed, a controllable input channel of LUT (an implementation of $\mathbb{G}$) is considered as an output channel from the viewpoint of a test case since the latter uses this channel to stimulate the LUT and conversely. On the other hand, input channels for the test case are both the output channels and uncontrollable channels of LUT. This is because, input values on uncontrollable channels will be provided by other subsystems, considered to be internal in the view of the overall system of LUT.
The set of variables of test case A^{tp} contains initially a global maximal waiting-time denoted as $WaitMax$, in order to bound the waiting delay for observing outputs or internal inputs emitted by other subsystems. The waiting-time fixed in the test case to avoid waiting for too long for such observations. For instance, it can be chosen as the maximal value among those which bound the occurrences of observations (if any) in the definition of the test purpose, plus one time unit. In addition, the set A^{tp} contains other variables that will be created during the construction of transitions in the test case.

For the TIOSTS test case $\mathbb{G}^{tp} = (Q^{tp}, q_0^{tp}, Tr^{tp})$, we have:

- Q^{tp} is the set of states containing states of test verdicts $Pass$, $Fail_{out}$, $Fail_{time}$, Inc_{out}, Inc_{time}, $Inc_{specIntI}$, $Inc_{uspecIntI}$ and other states constructed from the rules;
- q_0^{tp} is the initial state, here q_{ec_1}, where ec_1 is the initial context of tp;
- Tr^{tp} is the set of transitions built by applying a process that will be discussed in the following. This process will be applied to a so-called fully-prepared symbolic tree containing the test purpose tp. Figure 3.5 is a simplified illustration for this symbolic tree.

Figure 3.5 depicts a symbolic tree and a test purpose $tp = ec_1.ec_2.ec_3.ec_4.ec_5.ec_6$ (being as glimpsed before, a symbolic path in the tree). Besides the (execution) contexts of tp, the symbolic tree contains also the adjacent contexts outside of tp reflecting either a specified output (ec'_3), or an internal input received from another subsystem (may be ec'_5). All these contexts do not allow us to follow tp, hence, they do not represent the behavior

that we would like to observe. Adding these neighboring contexts in the tree, prepares for application of test case generation rules. Figure 3.5 is a simplified illustration, the details will be given in Section 3.5.2.

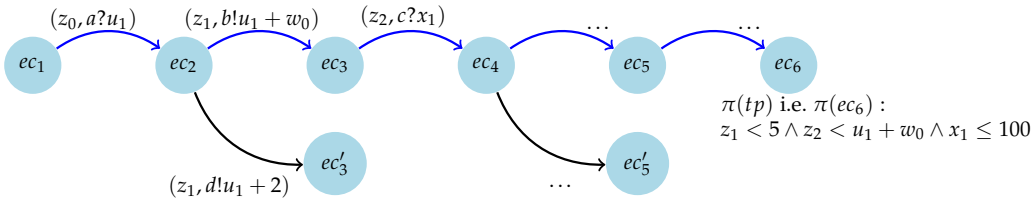


FIGURE 3.5: An example of fully-prepared symbolic tree for $tp : ec_1.ec_2.ec_3.ec_4.ec_5.ec_6$

For the given fully-prepared symbolic tree, the application process of our construction rules of the test case is defined as follows:

1. We advance linearly in the $tp = ec_1.ec_2.ec_3...ec_n$, starting with the context ec_2;
2. For the current ec, one applies all possible rules in the order in which they are presented;
3. In case where we can no longer apply any rules, then we move on to the next context in tp (becoming the new current ec) until we reach ec_n and finish with it.

In the following, we will not present all the contexts (ec'_3, ec'_5, ...) to facilitate the illustration. And some rules will be illustrated by an application example to facilitate their understanding, in addition to next Section 3.4 which is dedicated to the illustration for the ATM.

Let us note that a transition which is generated in test case will be presented in the form of a table for readability reason:

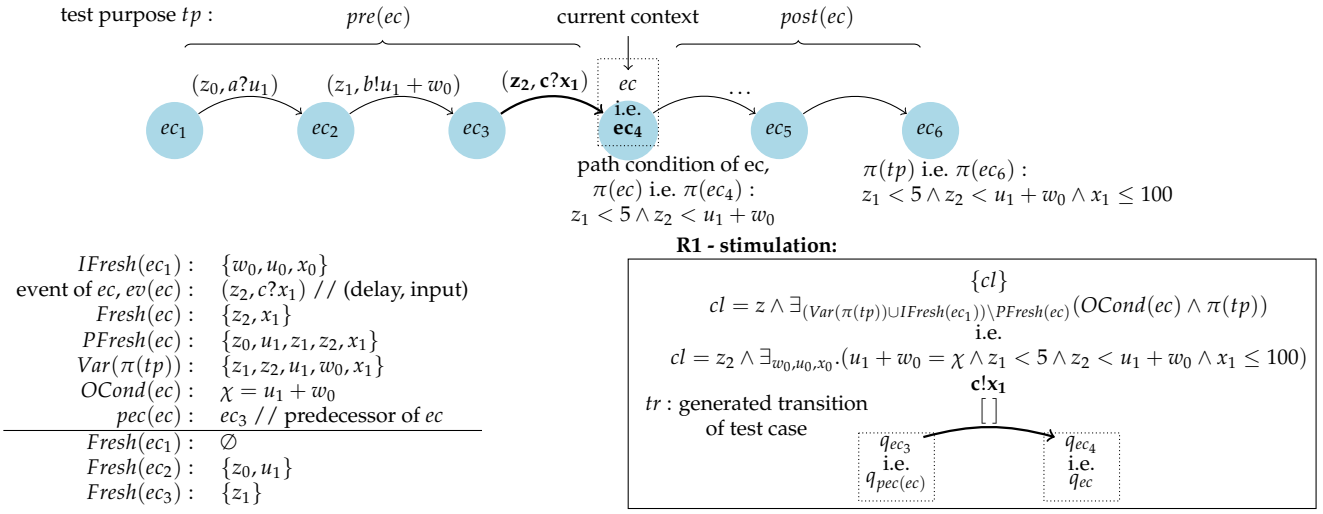
Source	**Clocks**	**Action**	**Substitution**	**Target**
q	$\mathbb{K}$	act	$[\,]$	q'
Guard				
ϕ				

where $[\,]$ is the identity substitution, it is used when the transition does not modify the values of variables. In our approach, for any given transition tr that will be generated in the test case, the test case does not need to update the values of variables, hence $\rho(tr) : [\,]$.

- **R1 - stimulation:** Case $ev(ec) = (z, c?x)$ with $c \in CC^{in}$, we build the following transition

Source	**Clocks**	**Action**	**Substitution**	**Target**
$q_{pec(ec)}$	$\{cl\}$	$c!x$	$[\,]$	q_{ec}
Guard				
$cl = z \wedge \exists_{(Var(\pi(tp)) \cup IFresh(ec_1)) \setminus PFresh(ec)} (OCond(ec) \wedge \pi(tp))$				

A stimulation in the test case is constructed in such a way that allows to follow the test purpose, therefore its guard is determined based on the path condition $\pi(tp)$ of the test purpose. A stimulation of the test case has to be emitted within a delay allowed by the test purpose, i.e. $cl = z$. Furthermore, the construction of the guard is based on identification conditions $OCond(ec)$ which have been previously accumulated during the test execution. The delay z and variable x are considered as variables in test case and therefore, will be added to the set of variables A^{tp}, since test cases need to know the moment and the value of stimulation. In order to illustrate the different ingredients of the rule, let us consider the following toy example.



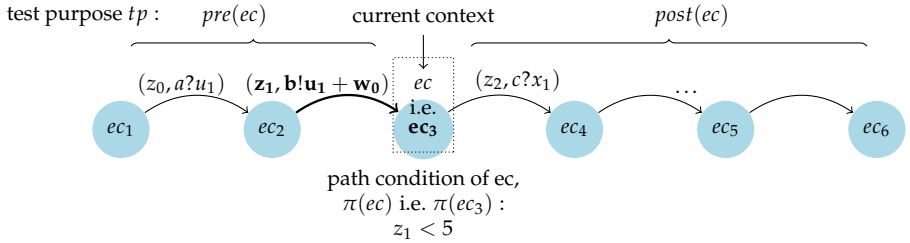
By interacting with the LUT, the values of some variables appearing in the PC will be progressively revealed, i.e., they will be associated with known values in test case as the execution progresses. The set $PFresh(ec)$ represents these variables: for example, the delay z_2 of the current execution context ec and its reception variable x_1 belong both to the set $PFresh(ec)$ ($z_2, x_1 \in PFresh(ec)$), as the test case knows when emitting a stimulation and its value. This holds as well for previously encountered variables before reaching ec ($z_0, u_1, z_1 \in PFresh(ec)$). Now to define the guard of the transition of the test case, one needs to satisfy $\pi(tp)$ while keeping unconstrained all non-revealed variables appearing in $\pi(tp)$ and initialized variables in $IFresh(ec_1)$ (cf remark 2.3.1), those are determined by the set resulting from excluding from the union $Var(\pi(tp)) \cup IFresh(ec_1)$ all variables that have been revealed during test execution to this level, i.e., represented by the set $PFresh(ec)$ (in the example $(Var(\pi(tp)) \cup IFresh(ec_1)) \setminus PFresh(ec) = \{w_0, u_0, x_0\}$). The action of stimulation is determined by a *mirror operator*: an input in the model becomes an output (stimulation) in the test case, and conversely, an output in the model becomes an input (reception) in the test case. In the example, the stimulation of the current ec, $c?x_1$, is transformed into the output $c!x$ in the test case.

- **R2 - authorized reaction inside** tp: Case $ev(ec) = (z, c!t)$,

	Source	Clocks	Action	Substitution	Target
if $post(ec) \neq \epsilon$	$q_{pec(ec)}$	$\{cl\}$	$c?OFresh(pec(ec))_c$	$[\,]$	q_{ec}
else		$\emptyset$			$Pass$
	Guard				
	$cl < WaitMax \wedge cl = z \wedge \exists_{IFresh(ec_1)\setminus PFresh(ec)}(OCond(ec) \wedge \pi(ec))$				

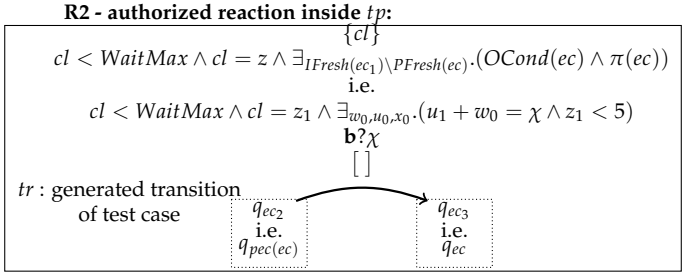
N.B., we denote $\widetilde{\phi}(tr) = \neg(\exists_{IFresh(ec_1)\setminus PFresh(ec)}(OCond(ec) \wedge \pi(ec)))$. The construction of $\widetilde{\phi}(tr)$ is necessary for constructing transitions by R4.
We recall that $OFresh(pec(ec))_c$ denotes here the (same) fresh variable used to represent from context $pec(ec)$, the received values on channel c for all its successor contexts (including ec, see Remark 2.3.1). The delay z and fresh variable $OFresh(pec(ec))_c$ are added to the set of variables A^{tp} of the test case since delay z is revealed (moment of reception) and $OFresh(pec(ec))_c$ is created by the test case to store observable values.

Let us consider again the same toy example, yet with another decomposition of tp with respect to an output event in order to illustrate the rule.



$IFresh(ec_1)$: $\{u_0, w_0, x_0\}$
event of ec, $ev(ec)$: $(z_1, b!u_1 + w_0)$ // (delay, output)
$Fresh(ec)$: $\{z_1\}$
$PFresh(ec)$: $\{z_0, u_1, z_1\}$
$OCond(ec)$: $\chi = u_1 + w_0$
$pec(ec)$: ec_2 // predecessor of ec
$OFresh(pec(ec))_b$: χ

R2 - authorized reaction inside tp:



The expected reaction $(b!u_1 + w_0)$ of LUT on channel b is seen as an input (b?χ) from the viewpoint of the test case, which leads to the creation of a new fresh variable $(OFresh(pec(ec))_b = \chi)$ to store the value received on that channel.
Let us discuss how the guard of an authorized reaction has been defined. First, we require that the test case waits for reactions during a time delay bounded by a parameter bound $WaitMax$ ($cl < WaitMax$), which is of practical concerns as this duration can be unconstrained and can arrive after an arbitrary long delay (as it is the case in this example). As previously, the guard takes into consideration the identification for the variables $OCond(ec) : \chi = u_1 + w_0$ and for the clock $cl = z_1$, where z_1 is the delay elapsed since the last event (hence the utility of clock resets by generated transitions of the test case). This delay is then verified to be compliant with path condition $\pi(ec)$ (here $z_1 < 5$, constrains its emission delay to be less than 5 time units). One can see that values accepted by the test case and received on χ are exactly those satisfying $cl = z_1 \wedge \exists_{w_0,u_0,x_0}.(\chi = u_1 + w_0 \wedge z_1 < 5)$, u_1 is supposed to be known as being a stimulus sent by the test case (its corresponding transition is generated by applying rule R1 on ec_2). Therefore, the satisfiability of the guard allows the test case to remain within the test purpose tp, and the test case evolves from state $q_{pec(ec)}$ to state q_{ec}. Finally, the clock cl is reset to measure the time elapsed for the next event.

In case where $post(ec)$ is empty ($post(ec) = \epsilon$), the generated transition is the one which allows to reach the verdict state $Pass$, meaning that the behavior of LUT is correct w.r.t the selected test purpose. Then, the clock cl does not need to be reset.

- **R3 - authorized reaction outside tp:**

  In case there exists $ec' \neq ec$ such that $pec(ec') = pec(ec)$ and $ev(ec') = (z, c!t)$,

Source	Clocks	Action	Substitution	Target
$q_{pec(ec)}$	$\emptyset$	$c?OFresh(pec(ec))_c$	$[\,]$	Inc_{out}
Guard				
$cl < WaitMax \wedge cl = z \bigwedge \exists_{IFresh(ec_1)\setminus PFresh(ec')}(OCond(ec') \wedge \pi(ec'))$				

N.B., we denote $\widetilde{\phi}(tr) = \neg(\exists_{IFresh(ec_1)\setminus PFresh(ec')}(OCond(ec') \wedge \pi(ec')))$. The construction of $\widetilde{\phi}(tr)$ is necessary for constructing transitions by R4.

We consider here the case when there exists an execution context ec' satisfying the following conditions: i) ec (the current context of tp w.r.t. our application process of construction rules) and ec' (distinct from ec) have a same predecessor, and ii) the event of ec', i.e., $ev(ec') = (z, c!t)$, is an output event. Then the action of the corresponding transition in the test case is an input, i.e. $c?\chi(ec)_c$ (dual of $c!t$) as in rule R2. The guard is defined as well is a similar manner as rule R2: yet the event needs to be compliant with path condition $\pi(ec')$. Now as this a specified output in the model, yet divergent outside tp, the verdict Inc_{out} is emitted and execution of the test case is ended. Once again, similar to R2, the delay z and fresh variable $OFresh(pec(ec))_c$ are added to the set of variables A^{tp} of test case.

This rule can be applied to the context ec'_3 that represents a specified behavior in the model in Figure 3.5. However, this reaction does not allow test case to follow the test purpose tp.

- **R4 - unauthorized reaction:**

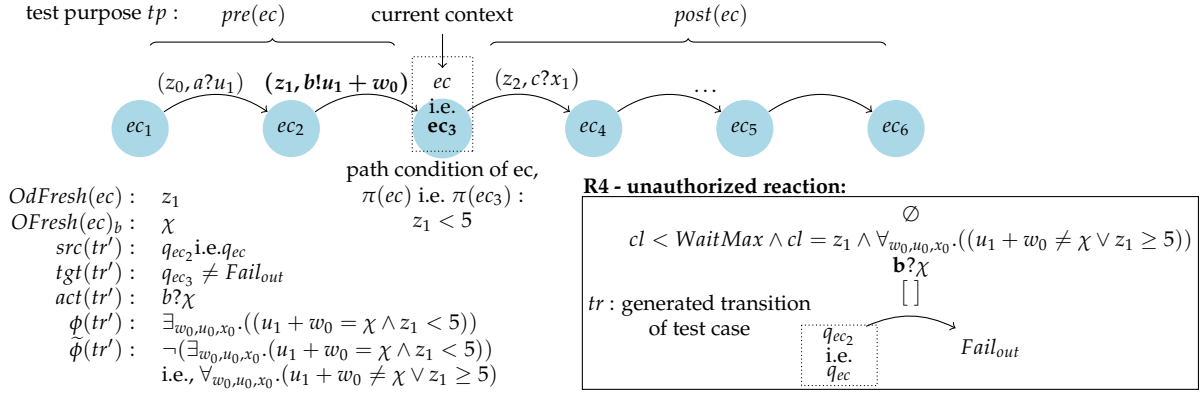
Source	Clocks	Action	Substitution	Target
q_{ec}	$\emptyset$	$c?OFresh(ec)_c$	$[\,]$	$Fail_{out}$
Guard				
$cl < WaitMax \wedge cl = OdFresh(ec) \bigwedge_{tr' \in Tr'} \widetilde{\phi}(tr')$				

Where Tr' is the set $\{tr' \mid tr' \in Tr^{tp} \wedge src(tr') = q_{ec} \wedge tgt(tr') \neq Fail_{out} \wedge act(tr') = c?OFresh(ec)_c\}$ and $\widetilde{\phi}(tr')$ is the formula defined in R2 and R3.

In case where there are authorized reactions tr' on a same output channel c: the unauthorized behaviors are constructed according to the authorized ones on the channel c. Indeed, those are characterized by all transitions tr' in the test case satisfying the following conditions: i) their source state must be the same whereas their target state must be different from $Fail_{out}$, and ii) their action is an input on the channel c.
Now, in order to define the erroneous received values and and/or their erroneous delays, we reason on the negation of a sub formula of $\phi(tr')$ (See rules R2 and R3 defining authorized reactions), denoted by $\widetilde{\phi}(tr')$: This formula is obtained by: i) omitting $cl < WaitMax \wedge cl = z$ in $\phi(tr)$ as it will be factorized (same

as $cl < WaitMax \wedge cl = OdFresh(ec)$, knowing that $OdFresh(ec) = z$), ii) and especially by negating the conditions defining specified values of reactions and their accepted delays in $\phi(tr')$.
A transition leads to $Fail_{out}$ if and only if the test case receives a reaction from the LUT on the channel c and either the corresponding delay or the data received on $OFresh(ec)_c$ does not satisfy the conjunction of all $\widetilde{\phi}(tr')$, i.e. meaning that there is no guard of an authorized reaction (within tp or outside tp) which is satisfiable.
The reception of such reaction leads to an authorized behavior of the LUT, denoted by the verdict $Fail_{out}$. Finally, the clock cl is reset to measure the time elapsed for the next event. We note that we do not need to add the delay $OdFresh(ec)$ and fresh variable $OFresh(ec)_c$ to the set A^{tp} of test case because they have been already as a application of R2 and R3 to generate specified reactions.
In case there is no authorized reaction on an output channel c (hence no corresponding transition tr'): the guard of the unauthorized reaction on c is reduced to $cl < WaitMax \wedge cl = OdFresh(ec)$, where $WaitMax$ denotes a maximum delay that the test case can wait for the observation, which will be erroneous when observed (the LUT is not supposed to react).

Let us consider again the toy example with the objective to illustrate the definition of transitions leading to the $Fail_{out}$ state in the test case. The unauthorized reaction is constructed based on the authorized transition constructed by rule R2 and denoted by tr'. Suppose that the test case receives a value that is stored to the new fresh variable χ after a duration z_1. A reaction on channel b is considered as unauthorized if either the value stored in χ is not the expected one or the delay z_1 does not satisfy the constraint of $\phi(tr')$. Therefore, the guard of transition for unauthorized reaction is defined by $\widetilde{\phi}(tr')$, which is obtained by negating the conditions constructed on specified values and accepted delays in $\phi(tr')$. Finally, we have $\phi(tr) : cl < WaitMax \wedge cl = z_1 \wedge \forall_{w_0,u_0,x_0}.((u_1 + w_0 \neq \chi \vee z_1 \geq 5))$.



- **R5 - internal input inside tp:** Case $ev(ec) = (z, c?x)$ and $c \in UC^{in}$, we build the following transition

	Source	Clocks	Action	Substitution	Target
if $post(ec) \neq \epsilon$	$q_{pec(ec)}$	$\{cl\}$	$c?OFresh(pec(ec))_c$	[ ]	q_{ec}
else		$\emptyset$			$Pass$
	Guard				
	$cl < WaitMax \wedge cl = z \bigwedge \exists_{IFresh(ec_1) \setminus PFresh(ec)}(OCond(ec) \wedge \pi(ec))$				

N.B., we denote $\widetilde{\phi}(tr) = \neg(\exists_{IFresh(ec_1) \setminus PFresh(ec)}(OCond(ec) \wedge \pi(ec)))$. The construction of $\widetilde{\phi}(tr)$ is necessary for constructing transitions by R7.

We remind the reader that $OFresh(pec(ec))_c$ is x (see Remark 2.3.1).

Let consider an execution context ec whose delay is z and action is an input on the channel c where c is an uncontrollable channel. The test case does not send this input, but it is able to observe this internal input which is sent by another subsystem, hence the action of tr is $c?OFresh(pec(ec))_c$.
The guard is defined based on the path condition $\pi(ec)$. Indeed, the reception of this input by the LUT needs to be compatible with the test purpose in order to follow targeted execution. In case of incompatibility, we are either in the case of an internal input outside of the test purpose or under-specified internal input, that will be introduced in the next two rules. As for reactions, the waiting-time for internal inputs is constrained by the bound $WaitMax$, i.e. $cl < WaitMax$. The delay z and fresh variable $OFresh(pec(ec))_c$ for storing observable values are added to the set A^{tp} of test case.

- **R6 - internal input outside tp:** In case there exists $ec' \neq ec$ such that $pec(ec') = pec(ec)$ and $ev(ec') = (z, c?x)$, $c \in UC^{in}$:

Source	**Clocks**	**Action**	**Substitution**	**Target**
$q_{pec(ec)}$	$\emptyset$	$c?OFresh(pec(ec))_c$	$[\,]$	$Inc_{specIntI}$
Guard				
$cl < WaitMax \wedge cl = z \bigwedge \exists_{IFresh(ec_1)\setminus PFresh(ec')}(OCond(ec') \wedge \pi(ec'))$				

N.B., we denote $\widetilde{\phi}(tr) = \neg(\exists_{IFresh(ec_1)\setminus PFresh(ec')}(OCond(ec') \wedge \pi(ec')))$. The construction of $\widetilde{\phi}(tr)$ is necessary for constructing transitions by R7.

As in previous rule, we remind the reader that $OFresh(pec(ec))_c$ is x (see Remark 2.3.1).

This rule applies on an execution context ec' satisfying the following conditions: i) its predecessor is the same as ec (the current context of tp w.r.t. our application process of construction rules), and ii) its action is of the form $c?x$, being an internal input sent by another subsystem. In such situation, the internal input is not expected by the test purpose, yet specified in the model. The guard construction is similar to the case of internal input inside of the test purpose (see Rule 5), when satisfied the testing is stops and the verdict $Inc_{specIntI}$ is emitted. The delay z and fresh variable $OFresh(pec(ec))_c$ are added to the set A^{tp} of test case.

- **R7 - under-specified internal input:**

Source	**Clocks**	**Action**	**Substitution**	**Target**
q_{ec}	$\emptyset$	$c?OFresh(ec)_c$	$[\,]$	$Inc_{uspecIntI}$
Guard				
$cl < WaitMax \wedge cl = OdFresh(ec) \bigwedge_{tr' \in Tr'} \widetilde{\phi}(tr')$				

Where Tr' is the set $\{tr' \mid tr' \in Tr^{tp} \wedge src(tr') = q_{ec} \wedge tgt(tr') \neq Inc_{uspecIntI} \wedge act(tr') = c?OFresh(ec)_c\}$ and $\widetilde{\phi}(tr')$ is the formula defined in R5 and R6.

Let us consider all the transitions in the test case tr' satisfying the following conditions: i) their source state is q_{ec}, and ii) their action is an a specified internal input (either inside or outside the test purpose). The firing

guard of the transition tr, i.e., conditions on received internal input values and their delay which are not in the model, is then defined over the negation formulas $\widetilde{\phi}(tr')$. When such conditions are satisfied, the verdict $Inc_{uspecIntI}$ is issued by the test case. The current construction is somewhat similar to the situations of unspecified outputs (see rule R4, when the verdict $Fail_{out}$ is emitted), it is simply transposed to internal inputs. The delay $OdFresh(ec)$ and fresh variable $OFresh(ec)_c$ are not added to the set A^{tp} of test case which contains already them from the application of R5 and R6.

- **R8 - authorized quiescence on bound $WaitMax$ ($WaitMax$ is the max. waiting-time for observations)**: Case ec such that there exists ec' satisfying $pec(ec') = ec$ and $chan(ec) \in (C^{out} \cup UC^{in})$,

Source	**Clocks**	**Action**	**Substitution**	**Target**
q_{ec}	$\emptyset$	_	$[\,]$	Inc_{time}
Guard				
$cl \geq WaitMax \wedge cl = OdFresh(ec) \bigwedge(\phi_\delta \vee \phi_{out} \vee \phi_{intI})$				

N.B., $\phi_\delta = \bigwedge_{ec' \in React(ec)}(\forall_{IFresh(ec_1)\setminus PFresh(ec)}(\forall_z(\neg(OCond(ec) \wedge \pi(ec')))))$, where $z = OdFresh(ec)$ ($OdFresh(ec)$ is the fresh duration introduced to represent the time elapsed from ec, see Remark 2.3.1), and finally $React(ec) = \{ec' \mid pec(ec') = ec \wedge chan(ec') \in C^{out}\}$ (see Definition 2.3.2),
$\phi_{out} = \bigvee_{ec' \in React(ec)} \exists_{IFresh(ec_1)\setminus PFresh(ec)}((OCond(ec) \wedge \pi(ec')))$,
$\phi_{intI} = \bigvee_{ec' \in Uncont(ec)} \exists_{IFresh(ec_1)\setminus PFresh(ec)}((OCond(ec) \wedge \pi(ec')))$,
where $Uncont(ec) = \{ec' \mid pec(ec') = ec \wedge chan(ec') \in UC^{in}\}$

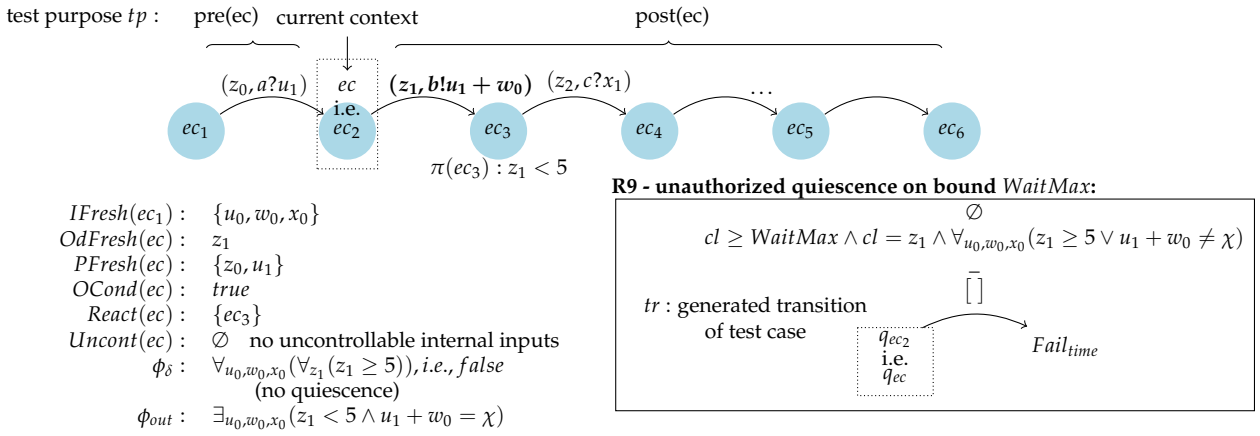
While waiting for observations (outputs or uncontrollable internal inputs), test case can stop by observing a quiescent situation of the the LUT, i.e., absence of action of the form $(z, \_)$. This situation is possible: i) either because it is authorized by the model (by quiescence enrichment, see Definition 2.3.2), or ii) because the test case waits long enough for observations and ends the test (the bound $WaitMax$ is reached, $cl \geq WaitMax$). Then, a special verdict Inc_{time} is emitted.

- **R9 - unauthorized quiescence on bound $WaitMax$ ($WaitMax$ is the max. waiting-time for observations)**: Case ec such that exists ec' satisfying $pec(ec') = ec$ and $chan(ec) \in (C^{out} \cup UC^{in})$,

Source	**Clocks**	**Action**	**Substitution**	**Target**
q_{ec}	$\emptyset$	_	$[\,]$	$Fail_{time}$
Guard				
$cl \geq WaitMax \wedge cl = OdFresh(ec) \bigwedge(\neg\phi_{out} \wedge \neg\phi_{intI})$				

we remark that $\neg\phi_\delta = \bigvee_{ec' \in React(ec)}(\exists_{IFresh(ec_1)\setminus PFresh(ec)}(\exists_z(OCond(ec) \wedge \pi(ec'))))$ captures exactly the situation when a duration is compatible with a reaction which is still possible to come, it is omitted as it contradicts the targeted verdict here, fail due to non-conforming duration.

The generated transition is enabled when the observed duration is not allowed in the model, and naturally when the guard of the transition defined by the previous rule R8 is not satisfied. The verdict $Fail_{time}$ is emitted in this case. In the following illustration, we show how the rule applies and generates a transition which captures the situation of non-conforming duration ($z_1 \geq 5$).



3.4 Illustration with ATM

Let us consider the following test purpose of the ATM:

$tp_1 = ec_1 \,.\, ec_2 \,.\, ec_3 \,.\, ec_4 \,.\, ec_5 \,.\, ec_6$ where each execution context is defined as follows:

- $ec_1 = (a_0, true, [rid := rid_0, amt := amt_0, wclock := 0, \ldots], (0, \_), ec_1)$
- $ec_2 = (a_1, true, [rid := 0, wclock := z_0], (z_0, \text{Init?}), ec_1)$
- $ec_3 = (a_2, true, [amt := amt_1, rid := 1, wclock := 0], (z_1, \text{Wdral?}amt_1), ec_2)$
- $ec_4 = (a_3, z_2 \leq 1, [wclock := z_2], (z_2, \text{Debit!}(1, amt_1 + fee_0, 1)), ec_3)$
- $ec_5 = (a_4, z_2 + z_3 < 4 \wedge z_2 \leq 1, [stat := stat_2, rid\_ret := rid\_ret_2, mid\_ret := mid\_ret_2, wclock := z_2 + z_3], (z_3, \text{Auth?}(rid\_ret_2, stat_2, mid\_ret_2)), ec_4)$
- $ec_6 = (a_1, mid\_ret_2 = 1 \wedge rid\_ret_2 = 1 \wedge stat_2 = \text{ACCEPT} \wedge z_2 + z_3 < 4 \wedge z_2 \leq 1, [stat := \text{ACCEPT}, wclock := 0], (z_4, \text{Cash!}amt_1), ec_5)$

The test purpose tp_1 describes the following behavior of the ATM: The ATM receives an initialization message on internal channel Init from the Bank to start its services. Then, it receives a withdrawal request on channel Wdral from a client, which is lately transferred to the Bank on internal channel Debit. Afterwards, the ATM machine receives the withdrawal authorization on internal channel Auth that allows it to return cash to the client on channel Cash. We recall that the set of output and input channels of ATM is respectively defined as $C^{out}(\text{ATM}) = \{\text{Abort, Cash, Debit, Log}\}$ and $C^{in}(\text{ATM}) = \{\text{Wdral, Init, Auth}\}$, whereas the set of controllable and uncontrollable input channels is respectively defined as $CC^{in}(\text{ATM}) = \{\text{Wdral}\}$ and $UC^{in}(\text{ATM}) = \{\text{Init, Auth}\}$. A controllable input channel is used to receive messages from the test case and an uncontrollable one is used to receive messages from other subsystems via the network.

The path condition of tp_1 is $\pi(tp_1) : mid\_ret_2 = 1 \wedge rid\_ret_2 = 1 \wedge stat_2 = \text{ACCEPT} \wedge z_2 + z_3 < 4 \wedge z_2 \leq 1$. From this formula, we can determine the set $Var(\pi(tp_1))$ of variables appearing in $\pi(tp_1)$ (i.e. $\pi(ec_6)$)

$$Var(\pi(tp_1)) = \{mid\_ret_2, rid\_ret_2, stat_2, z_2, z_3\}$$

The chosen test purpose allows the illustration of the following rules: a stimulation (R1), an authorized reaction inside tp_1 (R2), an unauthorized reaction (R4), and an internal input inside tp_1 (R5).

R5 - internal input inside tp_1.

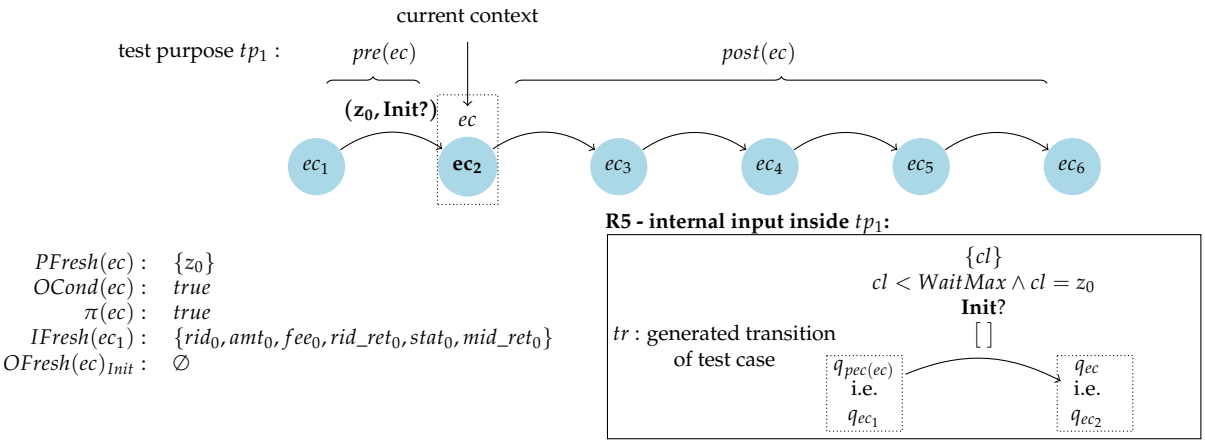


FIGURE 3.6: Internal input on the channel Init

We consider the decomposition $tp_1 = pre(ec_2) \,.\, ec_2 \,.\, post(ec_2)$ with $pre(ec_2) = ec_1$ and $post(ec_2) = ec_3 \,.\, ec_4 \,.\, ec_5 \,.\, ec_6$. The corresponding transition which will be generated in test case is denoted as tr. In the following, we construct different elements of tr.

Firstly, tr represents an observable input from the viewpoint of the test case since $act(ec_2) = \text{Init?}$ is an uncontrollable internal input issued by the Bank ($\text{Init} \in UC^{in}$). So, the test case is constructed in a way to observe this internal input, hence $act(tr) = \text{Init?}$. To wait for receptions, we constrain the waiting-time by the bound $WaitMax$. This internal input allows the test case to follow test purpose tp_1 if it is received during this bound $WaitMax$, i.e. $cl < WaitMax$ where cl is the clock used to measure the duration of transition $cl = z_0$. In addition, we need to take into consideration the path condition to ec_2 (i.e. $\pi(ec_2) = true$) and the identification condition to ec_2 (i.e. $OCond(ec_2) = true$, no values are transmitted via the channel Init) as the duration z_0 may be constrained by them. Therefore, $\phi(tr)$ is determined as $cl < WaitMax \wedge cl = z_0$. If the duration z_0 does not satisfy these constraints, the input is either an internal input outside of tp_1 yet in the model or under-specified, for both, an inconclusive verdict is emitted. Finally, the test case resets the clock cl in order to measure the duration of the next transition. The different elements of tr are shown in Figure 3.6.

R1 - stimulation on the channel Wdral.

In the following, we consider the decomposition $tp_1 = pre(ec_3) \,.\, ec_3 \,.\, post(ec_3)$ with $pre(ec_3) = ec_1 \,.\, ec_2$ and $post(ec_3) = ec_4 \,.\, ec_5 \,.\, ec_6$. The generated transition in test case is denoted as tr.

The considered context ec_3 is created by executing a transition labeled by an input action, thus the corresponding transition in the test case is labeled by an output (stimulation). For a stimulation, the test case needs to know the value and the moment of stimulation. In case of the considered transition, these are respectively defined as amt_1 and z_1. Now, for the purpose of following tp_1, this stimulation must be emitted by the test case within the delay allowed by the PC of test purpose (i.e. $\pi(tp_1)$). Furthermore, the value of stimulation must satisfy also $\pi(tp_1)$. Therefore, $\phi(tr)$ must take into account the constraint $cl = z_1 \wedge \pi(tp_1)$. In addition, the identification conditions which have been previously accumulated during the test execution may also constrain the value and the moment of stimulation. In case of ec_3, the identification condition $OCond(ec_3)$ is defined as $true$ since the test case has not received any reaction from the LUT yet. All non-revealed variables appearing in $\pi(tp_1)$ and initialized variables in $IFresh(ec_1)$ (i.e. $Var(\pi(tp_1)) \cup IFresh(ec_1) \setminus PFresh(ec_2)$) are existentially quantified since their values are unknown until this moment of stimulation. Therefore, $\phi(tr)$ is defined as follows:

$$cl = z_1 \wedge \exists_{(Var(\pi(tp_1)) \cup IFresh(ec_1)) \setminus PFresh(ec_2)} (mid\_ret_2 = 1 \wedge rid\_ret_2 = 1 \wedge stat_2 = \text{ACCEPT} \wedge z_2 + z_3 < 4 \wedge z_2 \leq 1)$$

with $Var(\pi(tp_1)) = \{mid\_ret_2, rid\_ret_2, stat_2, z_2, z_3\}$
and $IFresh(ec_1) = \{rid_0, amt_0, fee_0, rid\_ret_0, stat_0, mid\_ret_0\}$

Finally, the transition does not modify any variable, its substitution is denoted as $[\,]$ for the identity substitution.

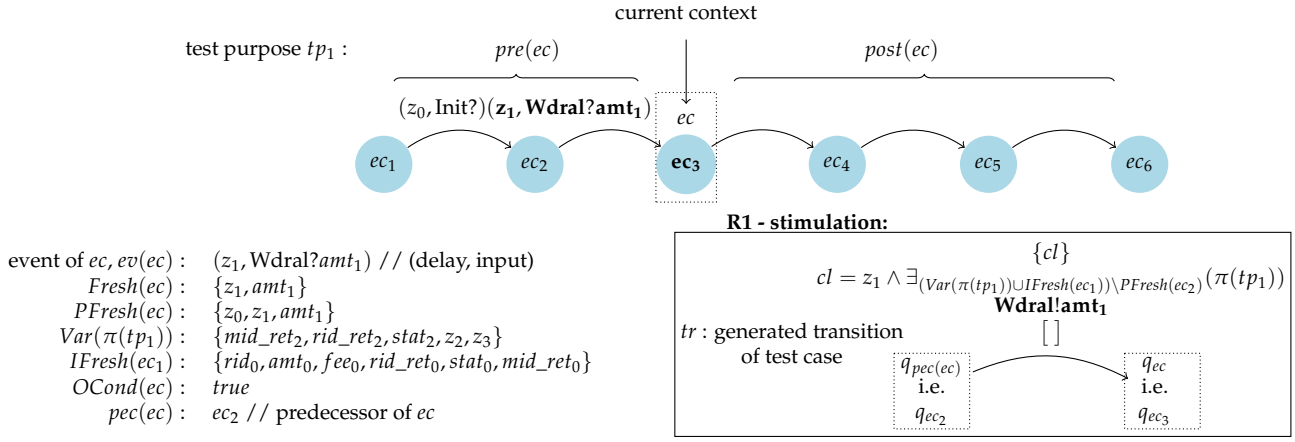


FIGURE 3.7: Stimulation on the channel Wdral

R2 - authorized reaction inside tp_1 on the channel Cash. In this illustration, we consider another decomposition of tp_1 into $pre(ec_6)\ .\ ec_6\ .\ post(ec_6)$ with $pre(ec_6) = ec_1 \ldots ec_5$ and $post(ec_6) = \epsilon$. This is an authorized reaction on the channel Cash.

Since ec_6 is reached by symbolic execution of a reaction on channel Cash from the LUT (Cash!amt_1), the corresponding transition tr from the viewpoint of the test case is labeled by a reception on this channel. Moreover, the reception of the value denoted by the symbol amt_1 on the channel Cash requires the creation of a new fresh variable χ_1^C in order to store this value [1]. Hence, the communication action is determined as Cash?(χ_1^C).

To define the guard $\phi(tr)$, first, we require that the test case waits for this reaction during a delay which is less than the bound $WaitMax$. In reality, the test case cannot wait for a very long delay to observe a reaction which can be unconstrained. The LUT's behavior conforms to the model if the reaction is received during an allowed delay z_1 which may be constrained by the condition $\pi(ec_6)$. Moreover, the construction of $\phi(tr)$ takes also into account the identification condition on the new fresh variables created to store received values, since they may be constrained by $OCond(ec_6)$. In case of ec_6, $OCond(ec_6)$ is determined as $OCond(ec_5) \wedge \chi_1^C = amt_1$ with $OCond(ec_5) : \chi_1^{D1} = 1 \wedge \chi_1^{D2} = amt_1 + fee_0 \wedge \chi_1^{D3} = 1$. As usual, we existentially quantify all non-revealed variables appearing in $OCond(ec_6) \wedge \pi(ec_6)$ until this reception (i.e. $IFresh(ec_1) \setminus PFresh(ec_6) = \{fee_0\}$ [2]). Thus, the guard $\phi(tr)$ is constructed as follows:

$$cl < WaitMax \wedge cl = z_4 \wedge \exists_{IFresh(ec_1) \setminus PFresh(ec_6)} (OCond(ec_6) \wedge \pi(ec_6))$$

with $OCond(ec_6) : \chi_1^{D1} = 1 \wedge \chi_1^{D2} = amt_1 + fee_0 \wedge \chi_1^{D3} = 1 \wedge \chi_1^C = amt_1$

[1] The subscript such as Ab, Au, C, D, L of new fresh variables for reception refers respectively to the channel Abort, Auth, Cash, Log, Debit

[2] To simplify, we present here only initialized variables which appear in the constraint $OCond(ec_6) \wedge \pi(ec_6)$.

and $\pi(ec_6) : mid\_ret_2 = 1 \wedge rid\_ret_2 = 1 \wedge stat_2 = \text{ACCEPT} \wedge z_2 + z_3 < 4 \wedge z_2 \leq 1$

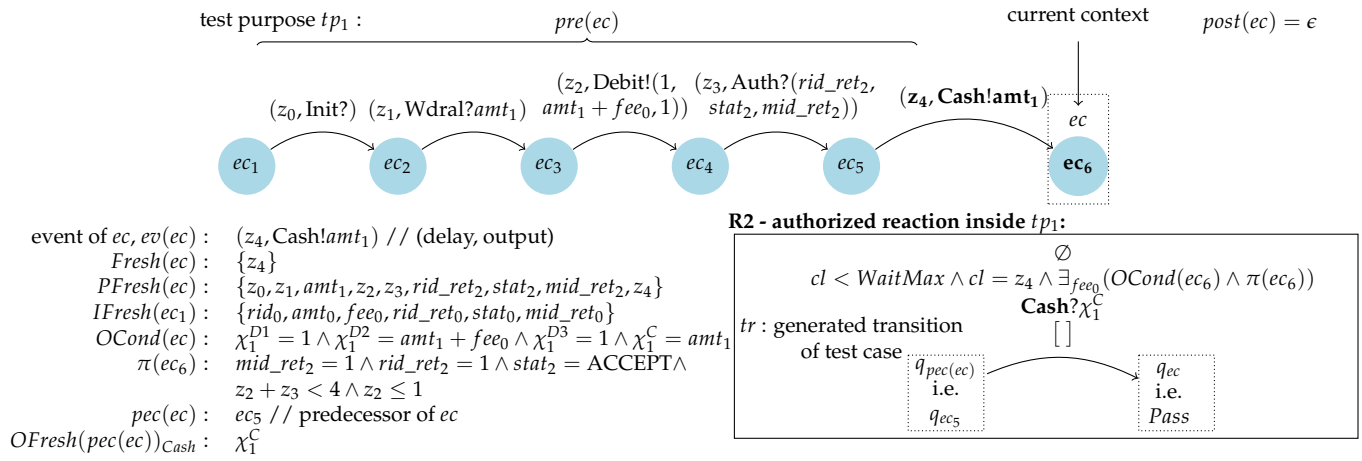


FIGURE 3.8: Authorized reception on the channel Cash

R4 - unauthorized reaction.

Let us discuss an unauthorized reaction or output on the channel Cash permitting to detect a non-conforming behavior of the LUT. Such erroneous reaction is determined either by receiving unspecified values on that channel or by observing a non-conforming delay of its occurrence. From the condition constructed on expected values and accepted delays in $\phi(tr')$ (i.e. $OCond(ec_6) \wedge \pi(ec_6)$) of the authorized reaction on the same channel, we can construct the guard of unauthorized transition by negating the constraint $OCond(ec_6) \wedge \pi(ec_6)$, expressing the reception of a non-conforming behavior in the delay $WaitMax$ (the observation waiting-time). Thus, the guard $\phi(tr)$ of the unauthorized reaction is constructed as follows:

$$cl < WaitMax \wedge cl = z_4 \wedge \neg(\exists_{fee_0}(OCond(ec_6) \wedge \pi(ec_6)))$$

with $OCond(ec_6) : \chi_1^D = 1 \wedge \chi_2^D = amt_1 + fee_0 \wedge \chi_3^D = 1 \wedge \chi_1^C = amt_1$
and $\pi(ec_6) : mid\_ret_2 = 1 \wedge rid\_ret_2 = 1 \wedge stat_2 = \text{ACCEPT} \wedge z_2 + z_3 < 4 \wedge z_2 \leq 1$
The elements of this transition are illustrated below:

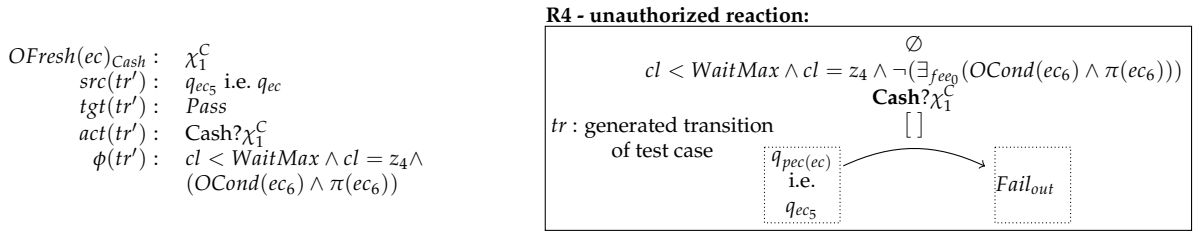


FIGURE 3.9: Unauthorized reception on the channel Cash

R9 - unauthorized quiescence on bound WaitMax.

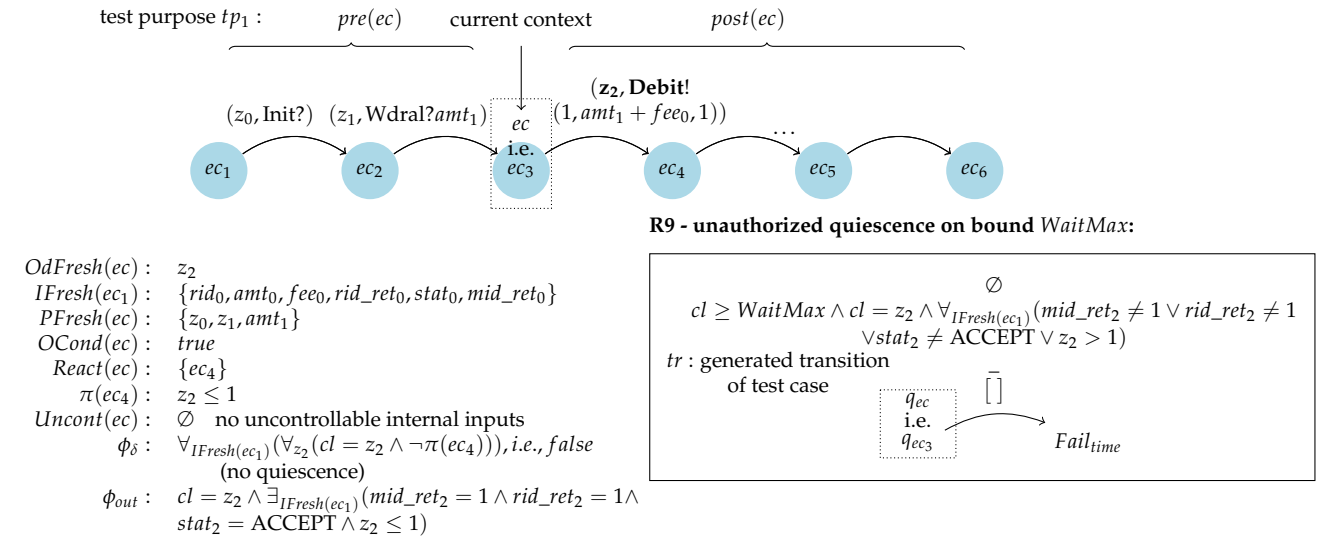


FIGURE 3.10: Unauthorized quiescence

The generated sequence transition for quiescence in test case represents the situation when the LUT does not supply a reaction as specified in the model. In case of the context ec_3, the system is supposed to reply on channel Debit within the duration $z_2 \leq 1$. In practice, the test case will wait for the reaction on channel Debit during this delay, and obviously the test case has already waited more than the predefined bound $WaitMax$. After that (i.e. $z_2 > 1$), the test case considers that the LUT is in quiescence. The verdict $Fail_{time}$ is emitted since the quiescence is not authorized in this situation, i.e. the test case is unable to provide the output on channel Debit within specified duration.

Let us illustrate in the figure below the form of the generated test case with several transitions.

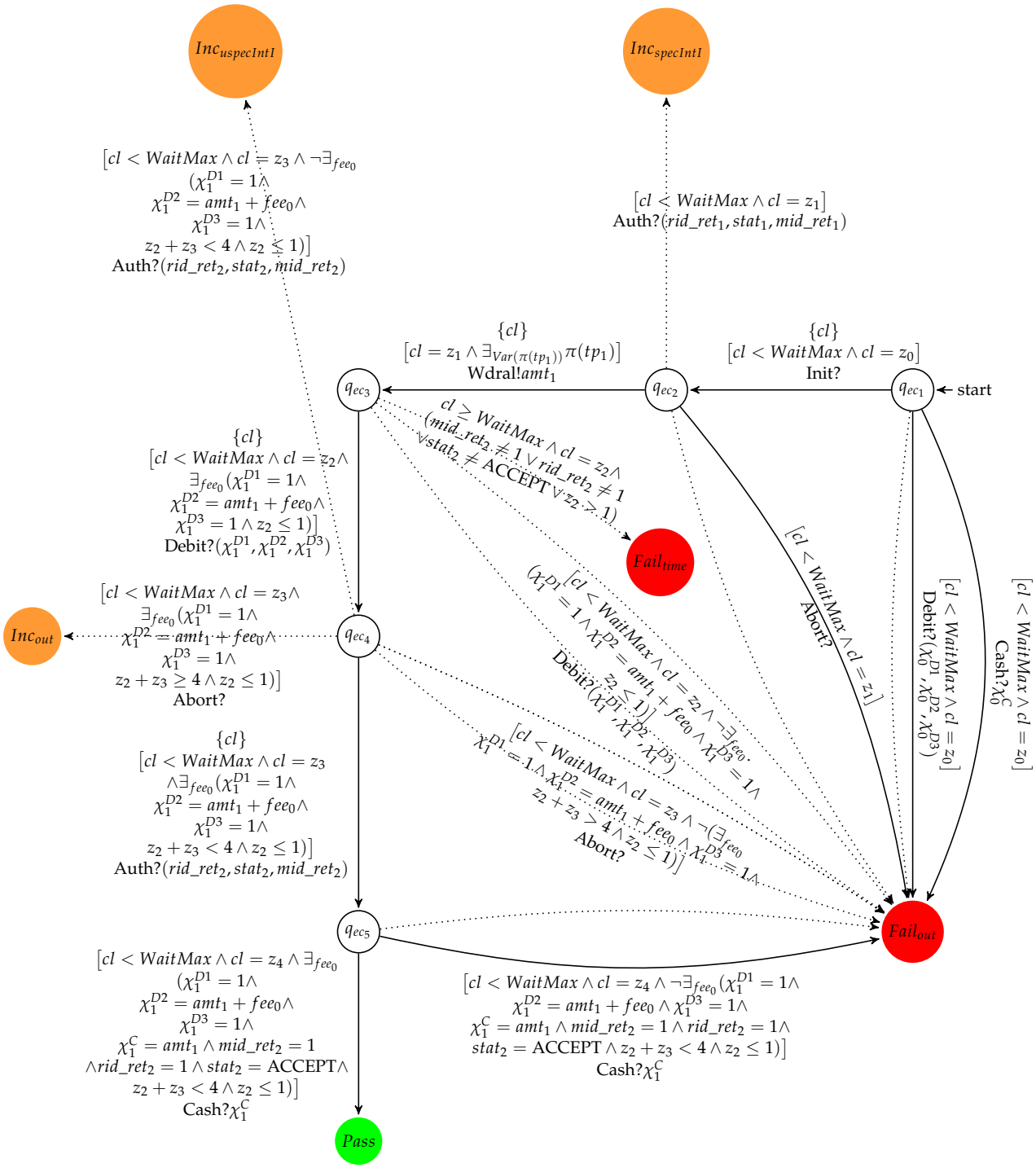


FIGURE 3.11: TIOSTS of the generated test case for ATM

Because of the big size of the test case, Figure 3.11 represents only its form with different verdict states and does not contain all transitions. For the sake of readability, the dotted arrow is used to denote some other transitions leading to different verdicts that we cannot detail due to lack of space.

3.5 Implementation in Diversity

Our testing approach for unitary systems is implemented in the Diversity tool. In this section, we are going to talk in detail about this implementation. In the first part, we start off by introducing the generic process of test case generation. The second part is dedicated to the implementation of a new module in Diversity to derive a test case from a feasible sequence of consecutive transitions.

3.5.1 Test case generation process

Given the model of a unitary system described by a TIOSTS $\mathbb{G}$ and a test purpose depicting a particular behavior in $\mathbb{G}$, the process takes as input the reference model $\mathbb{G}$ and a feasible sequence of consecutive transitions characterizing a test purpose, then produces a symbolic execution tree permitting to cover the selected test purpose.

1. Selection of a sequence of consecutive transitions in the TIOSTS $\mathbb{G}$ and use of the module TESTPURPOSE_SEL to verify if this sequence is feasible. If it is feasible, we move to step 2 as we can use the sequence to characterize a test purpose, otherwise we select another sequence of consecutive transitions;

2. Generation of a test case according to the test purpose which is characterized by the selected sequence of consecutive transitions from step 1;

3. Execution of the generated test case on the LUT with the goal to follow the test purpose.

For the purpose of illustrating this process, in the next subsection, let us present a new module called TESTCASE_GEN that has been implemented in Diversity and its purpose is to generate a test case from a given sequence of consecutive transitions.

3.5.2 Module of test case generation for unitary systems

In a formal way, the test purpose is defined as a symbolic path selected from the symbolic execution of $\mathbb{G}$ (see Section 3.3). However, the selection of a test purpose in the symbolic tree is not really convenient and suitable for Diversity's users while the symbolic tree becomes complex. For the sake of simplicity, we propose in our implementation a simpler way to select a test purpose: Instead of selecting directly the test purpose from the tree, we characterize it equally as a sequence of consecutive transitions in the model. Consequently, we can apply the module TESTPURPOSE_SEL that we have already implemented from Chapter 2 in order to verify the feasibility of the sequence of consecutive transitions. Then, our new module TESTCASE_GEN computes a symbolic tree covering this sequence and generate the test case associated with this sequence.

Figure 3.12 captures a screenshot of a workflow file in Diversity that illustrates the declaration of the module TESTCASE_GEN, as well as its parameters to generate the test case for ATM. It is noted that uncontrollable channels for ATM are also taken into account in the parameters, i.e. two channels `Init` and `Auth` are declared inside of the section `@uncontrollable`. Furthermore, users can parameterize location and name of the generated test case through the parameters `folder` and `file#1`.

```
TESTCASE_GEN 'test case generation for unitary systems' {
    Sequence [
        transition = "tr_1a"
        transition = "tr_2a"
        transition = "tr_3a"
        transition = "tr_4a"
        transition = "tr_5a"
    ]
    @uncontrollable [
        port = 'Init'
        port = 'Auth'
    ]
    vfs [
        folder = "testcase_generated"
        file#1 = "ATM.xlia"
    ] // end vfs
}
```

FIGURE 3.12: Screenshot from a workflow file of module TESTCASE_GEN for ATM

In Figure 3.13, we present the operation of module TESTCASE_GEN. In general, this module takes as input a TIOSTS and a feasible sequence of consecutive transitions *str* and outputs a test case generated associated with *str*. For this, TESTCASE_GEN computes a symbolic tree that covers the sequence and creates additional execution contexts which prepares for process of test case generation. Let us point out that this module intervenes in the steps **pre-filtering**, **post-filtering** and **post-processing** (see Section 2.4.2). In the following, we explain the mission of TESTCASE_GEN in each step.

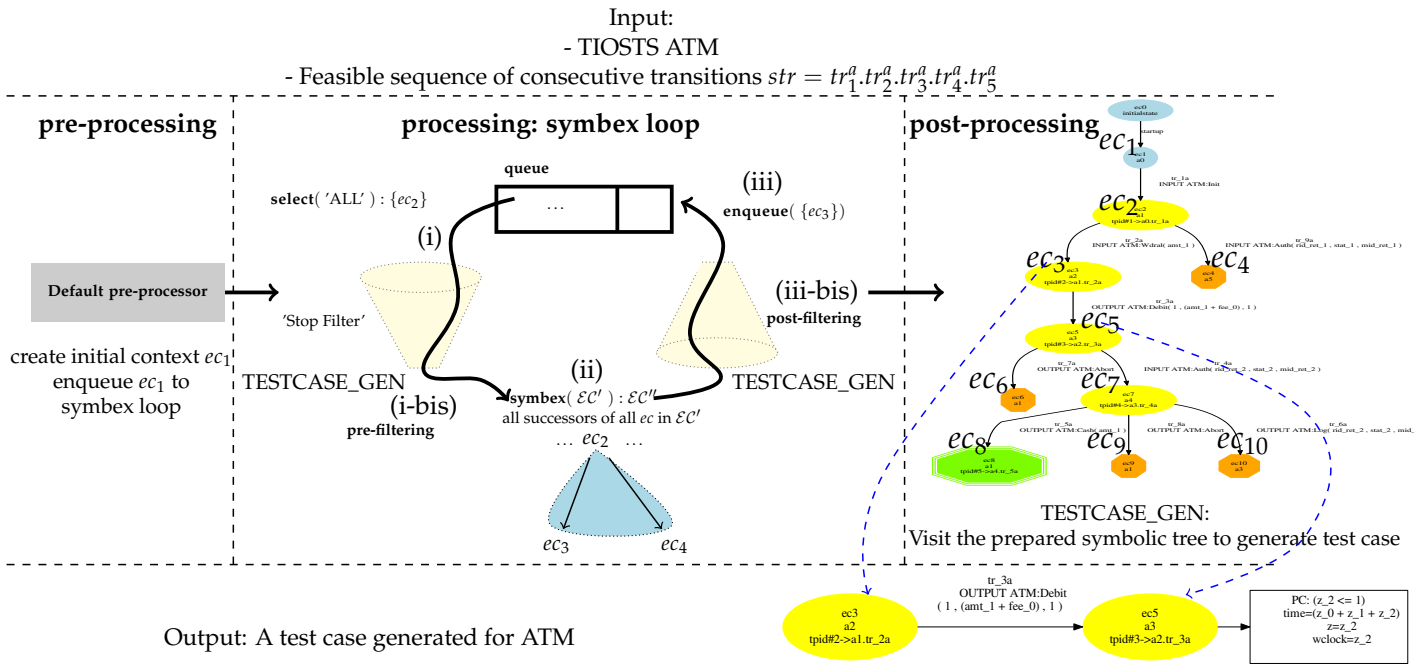


FIGURE 3.13: Module TESTCASE_GEN

In the step **pre-filtering**, the module verifies if the sequence coverage is completed. In that case, all contexts from the waiting queue are rejected and the symbex loop is stopped, otherwise the process continues. Let take the example in Figure 3.13, the context ec_2 is the only one in waiting queue and as the coverage is not completed, we compute successors of ec_2, which are determined as the following set $\{ec_3, ec_4\}$.

During the step **post-filtering**, the module checks the coverage whether it allows to cover the considered transition. In Figure 3.13, our module has covered so far the first transition tr_1^a. The step **post-filtering** allows us to filter ECs according to the transition in question. We find that ec_3 is filtered, then added to the symbolic tree and waiting queue since the execution leading to ec_3 has covered the transition in question tr_2^a, whereas the other one (ec_4) that has covered the transition tr_9^a is inserted to the symbolic tree. The adding of these contexts to the symbolic tree is useful in the process of test case generation in the step **post-processing**. Indeed, they represent specified behaviors in the model which do not allow to follow the behaviors that we would like to observe and therefore, will be evaluated as inconclusive verdict. Besides, our module also generates execution contexts representing the reception of internal actions sent by other subsystems which give rise to an inclusive verdict. In case where the sequence *str* is empty, the contexts for internal receptions are also generated and added to the symbolic tree. The symbolic tree that covers the test purpose is the fully-prepared symbolic tree that we presented in Section 3.3.

The fully-prepared symbolic tree obtained from the coverage of *str* is given in Figure 3.13. The following graphical conventions are applied to denote different kinds of contexts: the initial execution context ec_1 is colored in blue, an execution context permitting to cover the transitions in the sequence *str* is colored in yellow (ec_2, ec_3, ec_5, ec_7), except the one that covers the last transition is colored in green (ec_8), an execution context that either covers a transition which do not allow to follow the test purpose ($ec_4, ec_6, ec_9, ec_{10}$) or represents the reception of an internal action is colored in orange.

The step **post-processing** generates a test case from the fully-prepared symbolic tree by visiting all contexts in the tree to create for each of them a corresponding transition using the construction rules R1-R9 (Section 3.3) w.r.t. the type of action of the current visited context *ec*. For instance, in Figure 3.13, the generation process begins from the initial context ec_1 to the last one (i.e. ec_{10}). From the fully-prepared symbolic tree, the module TESTCASE_GEN generates, according to the construction rules, transitions that follows the test purpose from $ec_2, ec_3, ec_5, ec_7, ec_8$ (or leads to inconclusive verdict from $ec_4, ec_6, ec_9, ec_{10}$). To construct transitions leading to verdict *Fail* from any given state in the test case, we apply the construction rules R4 and R9.

For more information about the implementation of the module TESTCASE_GEN, reader is invited to visit the website: https://projects.eclipse.org/proposals/eclipse-formal-modeling-project

Let us illustrate in the following a transition generated by TESTCASE_GEN. We suppose that the module has already processed contexts ec_1, ec_2, ec_3 and ec_4 and now, it processes ec_5. From the pieces of information that contains the context ec_5 (cf. symbolic tree in Figure 3.13), the different elements of the corresponding transition in the test case are constructed.

The context ec_5 is the execution result of a transition labeled by an output, then the transition in test case is a reception, which creates a new communication channel Debit in the generated test case (Figure 3.14).

```
@com:
    connector< env >  {
        input ATM->Debit;
    }
```

FIGURE 3.14: Connection of channel Debit in the generated test case

New variables are created for receiving values emitted from the LUT (i.e. `chi_D1_3`, `chi_D2_3`, `chi_D3_3`). The transition is constrained by the following guard $cl\_TC < WaitMax \wedge cl\_TC = z_2 \wedge \exists_{\{mid\_ret_2, rid\_ret_2, stat_2, z_3\} \cap Fresh(ec_1)} (\texttt{chi\_D1\_3} = 1 \wedge \texttt{chi\_D2\_3} = amt_1 + fee_0 \wedge \texttt{chi\_D3\_3} = 1 \wedge z_2 \leq 1)$, conditioning the transition duration (i.e. $z_2 \leq 1$) and the values stored in the created variables `chi_D1_3`, `chi_D2_3`, `chi_D3_3`. The clock cl_TC is reset at the end of this transition in order to measure the duration of the next

transition. In Figure 3.15, we show a screenshot of some transitions in our generated test case from initial state q_1 to the verdict states $Pass$, $Fail_{out}$, Inc_{out}.

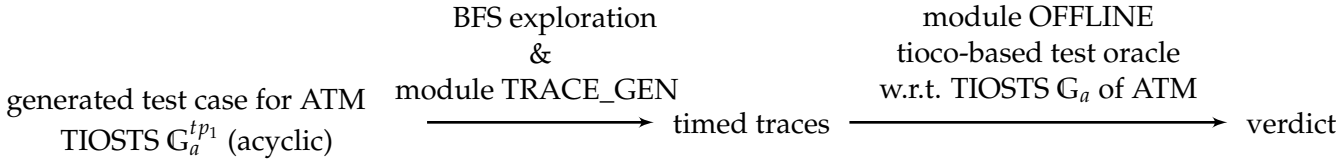
```
state q_ec1 {
    transition  --> q_ec2 {
        input Init;
        guard ((cl == z_0) && (cl < WaitMax));
        cl := 0;
    }// There are other transitions
}
state q_ec2 {
    transition  --> q_ec3 {
        guard ( (cl == z_1) && exists< mid_ret_0 : integer , rid_ret_0 : integer , stat_0 : Response ,
            rid_0 : integer , amt_0 : integer, fee_0 : integer, mid_ret_2 : integer ,
            rid_ret_2 : integer , stat_2 : Response , z_2 : urational , z_3 : urational >((mid_ret_2 == 1)
            && (rid_ret_2 == 1) && (stat_2 == ACCEPT) && ((z_2 + z_3) < 4) && (z_2 <= 1)));
        output Wdral(amt_1);
        cl := 0;
    }// There are other transitions
}
state q_ec3 {
    transition  --> q_ec4 {
        input Debit(chi_D1_1, chi_D2_1, chi_D3_1);
        guard ( (cl == z_2) && (cl < WaitMax) && exists< mid_ret_0 : integer , rid_ret_0 : integer ,
            stat_0 : Response , rid_0 : integer , amt_0 : integer, fee_0 : integer >((chi_D1_1 == 1) &&
            (chi_D2_1 == (amt_1 + fee_0)) && (chi_D3_1 == 1) && (z_2 <= 1)));
        cl := 0;
    }// There are other transitions
}
state q_ec4 {
    transition  --> q_ec5 {
        input Auth(rid_ret_2, stat_2, mid_ret_2);
        guard ( (cl == z_3) && (cl < WaitMax) && exists< mid_ret_0 : integer , rid_ret_0 : integer ,
            stat_0 : Response , rid_0 : integer , amt_0 : integer, fee_0 : integer >((chi_D1_1 == 1) &&
            (chi_D2_1 == (amt_1 + fee_0)) && (chi_D3_1 == 1) && (z_2 <= 1) && ((z_2 + z_3) < 4)) );
        cl := 0;
    }// There are other transitions
}
state q_ec5 {
    transition  --> Pass {
        input Cash(chi_C1_5);
        guard ((cl == z_4) && (cl < WaitMax) && exists< mid_ret_0 : integer , rid_ret_0 : integer ,
            stat_0 : Response , rid_0 : integer , amt_0 : integer, fee_0 : integer > ((chi_C1_5 == amt_1)
            && (chi_D1_1 == 1) && (chi_D2_1 == (amt_1 + fee_0)) && (chi_D3_1 == 1) && (mid_ret_2 == 1)
            && (rid_ret_2 == 1) && (stat_2 == ACCEPT) && ((z_2 + z_3) < 4) && (z_2 <= 1)));
    }
    transition  --> Fail_out {
        input Cash(chi_C1_5);
        guard ((cl == z_4) && (cl < WaitMax) && ( ! exists< mid_ret_0 : integer , rid_ret_0 : integer ,
            stat_0 : Response , rid_0 : integer , amt_0 : integer, fee_0 : integer > ((chi_C1_5 == amt_1)
            && (chi_D1_1 == 1) && (chi_D2_1 == (amt_1 + fee_0)) && (chi_D3_1 == 1) && (mid_ret_2 == 1)
            && (rid_ret_2 == 1) && (stat_2 == ACCEPT) && ((z_2 + z_3) < 4) && (z_2 <= 1))));
    }
    transition  --> Inc_out {
        input Log(chi_L1_5, chi_L2_5, chi_L3_5);
        guard ((cl == z_4) && (cl < WaitMax) && exists< mid_ret_0 : integer , rid_ret_0 : integer ,
            stat_0 : Response , rid_0 : integer , amt_0 : integer, fee_0 : integer >(((mid_ret_2 != 1) ||
            (rid_ret_2 != 1)) && (chi_D1_1 == 1) && (chi_D2_1 == (amt_1 + fee_0)) && (chi_D3_1 == 1) &&
            (chi_L1_5 == rid_ret_2) && (chi_L2_5 == stat_2) && (chi_L3_5 == mid_ret_2) &&
            ((z_2 + z_3) < 4) && ((z_2 + z_3 + z_4) <= 1) && (z_2 <= 1)));
    }
    transition  --> Inc_uspecIntI { |;|
        input Init;
        guard ((ATM.cl == z_4) && (cl < WaitMax));
    }// There are other transitions
}
```

FIGURE 3.15: An extract of the generated test case

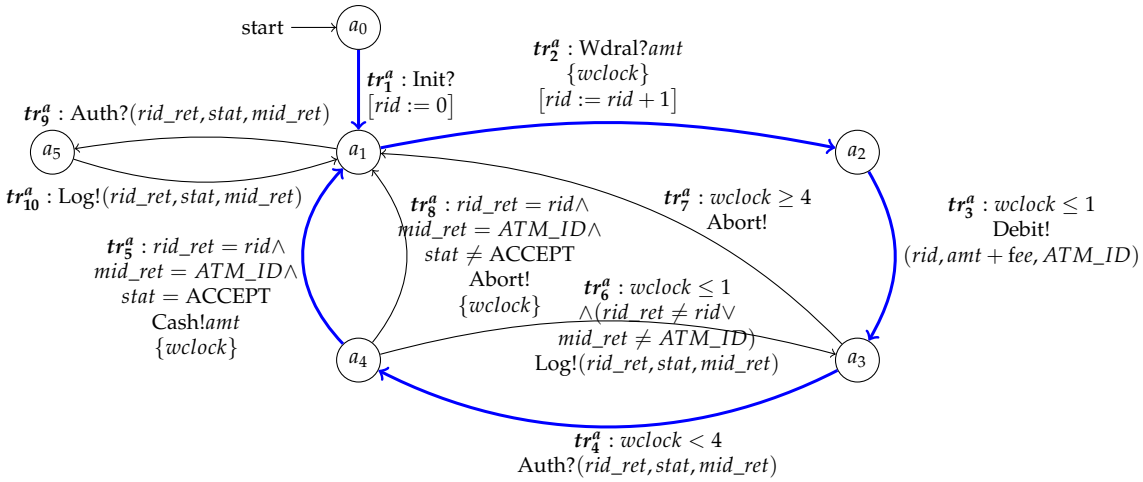
The module TESTCASE_GEN terminates its processing after taking into consideration the last context ec_{10}. Due to the size of the test case generated by our module TESTCASE_GEN, we will not detail it in the context of this chapter.

3.5.3 Elements on the validation of the generated test case

As previously mentioned, our test case is of the form TIOSTS, which is executable in the Diversity tool. In this part of the thesis, we propose some elements related to the validation of the generated test case presented in Subsection 3.4 by generating its timed traces. This step can be done by using the module traces generation in Diversity. Indeed, from a given symbolic tree, this module allows Diversity to generate a timed trace associated to each symbolic path in the tree. The purpose is to execute symbolically the TIOSTS test case and to apply the module TRACE_GEN to obtain timed traces of test case. Then, we verify the correctness of these traces thanks to the module OFFLINE implemented in Diversity [5] whose purpose is to compute a verdict on the correctness of a timed trace w.r.t the reference model of the ATM based on the conformance relation tioco. The presented steps can be summarized as follows:



Let us recall the ATM model which will service as a reference for verdict computation.



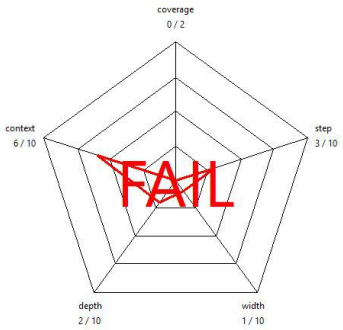
We consider the test purpose tp_1 which is characterized by the sequence of transitions colored by blue:

$$tr_1^a.tr_2^a.tr_3^a.tr_4^a.tr_5^a$$

.

First example of trace. Let us consider in the following the timed trace $\sigma_1 = (0, \text{Init?}).(0, \text{Wdral!}0).(0, \text{Debit?}(1,0,1)).(0, \text{Auth?}(0, \text{REJECT}, 0)).(0, \text{Cash?}0)$ of the test case which is generated by executing the sequence of transitions going through $q_{ec_1}, q_{ec_2}, q_{ec_3}, q_{ec_4}, q_{ec_5}$ and $Fail_{out}$ (see Figure 3.15). By applying the mirror operation, we obtain in the following the timed trace of the LUT $\sigma_1' = (0, \text{Init?}).(0, \text{Wdral?}0).(0, \text{Debit!}(1,0,1)).(0, \text{Auth?}(0, \text{REJECT}, 0)).(0, \text{Cash!}0)$.
This trace σ_1' is not specified by the reference model of ATM. Indeed, by applying the module OFFLINE to verify the conformance of σ_1' w.r.t to the TIOSTS ATM, we obtain the result in Figure 3.16 which shows the verdict *Fail*: the trace σ_1' is not accepted by ATM. Let us analyze in the following this result. The sequence of

the first, second and third event (respectively represented by $(0, \text{Init?})$, $(0, \text{Wdral?}0)$ and $(0, \text{Debit?}(1,0,1))$) belongs to the TIOSTS ATM and allows to follow the test purpose tp_1. We consider now the fourth event $(0, \text{Auth?}(0, \text{REJECT}, 0))$, describing a reception of the tuple $(0, \text{REJECT}, 0)$ by ATM on the uncontrollable channel Auth. Consequently, the variables *rid_ret*, *stat* and *mid_ret* are respectively assigned the value of 0, REJECT, 0. After receiving this tuple, the ATM machine reaches the state a_4. However, tr_5^a cannot be executed. Indeed, the values stored in $rid\_ret, stat, mid\_ret$ do not allow to execute tr_5^a since the guard $\phi(tr_5^a)$ is not satisfiable (i.e. $rid\_ret \neq rid, stat \neq$ ACCEPT and $mid\_ret \neq ATM\_ID$). The only possible transition which can be executed is tr_6^a, which gives us the verdict Inc_{out} as this reaction diverges from the test purpose tp_1. Therefore, the last event in σ_1' leads to the verdict $Fail_{out}$.

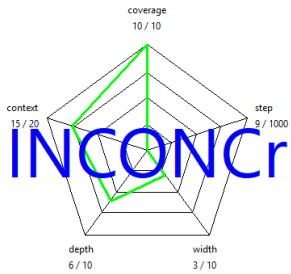



```
================================================================================
==========================  THE VERDICT IS : FAIL  ===========================
================================================================================
The smallest trace that has failed started with :
 trace#0< size: 10 > { |;|
  tpid#8 time:time inst::ATM_unitary.delay[ 0 ]
  tpid#9 com:output#env inst::ATM_unitary.ATM.Cash{ 0 }
 }
```

$$\sigma_1 = (0, \text{Init?}).(0, \text{Wdral!}0).\ (0, \text{Debit?}(1,0,1)).(0, \text{Auth?}(0, \text{REJECT}, 0)).(0, \text{Cash?}0)$$

FIGURE 3.16: Application of OFFLINE:
Trace unaccepted due to unspecified reaction on Cash

Second example of trace. In this example, we consider the second trace $\sigma_2 = (1/6, \text{Init?}).(1/6, \text{Wdral!}0).\ (1/6, \text{Debit?}(1,1,1)).(1/6, \text{Auth?}(0, \text{REJECT}, 0)).(1/6, \text{Log?}(0, \text{REJECT}, 0))$ obtained by executing the sequence of transitions from q_{ec_1}, q_{ec_2}, q_{ec_3}, q_{ec_4}, q_{ec_5} to Inc_{out}. This trace is equivalent to the following one for the LUT by doing mirror operation $\sigma_2' = (1/6, \text{Init?}).\ (1/6, \text{Wdral?}0).\ (1/6, \text{Debit!}(1,1,1)).\ (1/6, \text{Auth?}(0, \text{REJECT}, 0)).\ (1/6, \text{Log!}(0, \text{REJECT}, 0))$.
Figure 3.17 presents the result of application of OFFLINE to σ_2'. Let us explain this result as follows: The following prefix $(1/6, \text{Init?}).(1/6, \text{Wdral?}0).(1/6, \text{Debit!}(1,1,1)).(1/6, \text{Auth?}(0, \text{REJECT}, 0))$ of σ_2' is specified by the reference model, and the extension of this prefix by the event $(1/6, \text{Log!}(0,0,0))$ produced by the LUT gives us a specified behavior with respect to the reference model of the ATM machine. However, this event does not allow us to follow the test purpose tp_1 when the LUT has executed the transition tr_6^a (a reply on channel Log) instead of tr_5^a. The test case emits the verdict Inc_{out} for a specified reaction in the model, yet divergent from tp_1, even if the conformance relation tioco holds for this situation.

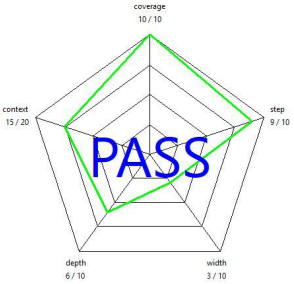



```
AVM OFF-LINE TEST  PROCESSOR
================================================================================
==========================  THE VERDICT IS : INCONCr  ==========================
================================================================================
```

$$\sigma_2' = (1/6, \text{Init?}).(1/6, \text{Wdral?}0).(1/6, \text{Debit!}(1,1,1)).(1/6, \text{Auth?}(0, \text{REJECT}, 0)).(1/6, \text{Log!}(0, \text{REJECT}, 0))$$

FIGURE 3.17: Application of OFFLINE:
Trace specified by the model but outside of test purpose

Third example of trace. We examine in the following the trace $\sigma_3 = (0, \text{Init?}).(0, \text{Wdral!}0).(0, \text{Debit?}(1,0,1)).(0, \text{Auth?}(1, \text{ACCEPT}, 1)).(0, \text{Cash?}0)$ of the test case. This trace is obtained from the sequence of transitions covering $q_{ec_1}, q_{ec_2}, q_{ec_3}, q_{ec_4}, q_{ec_5}$ and *Pass* in Figure 3.15. The equivalent trace in the LUT is the following $\sigma'_3 = (0, \text{Init?}).(0, \text{Wdral?}0).(0, \text{Debit!}(1,0,1)).(0, \text{Auth?}(1, \text{ACCEPT}, 1)).(0, \text{Cash!}0)$.
Once again, the correctness of σ'_3 is verified by using the module OFFLINE (Figure 3.18). This trace is a conforming behavior of LUT w.r.t to the TIOSTS ATM. It is straightforward to find that the prefix $(0, \text{Init?}).(0, \text{Wdral?}0).(0, \text{Debit!}(1,0,1)).(0, \text{Auth?}(1, \text{ACCEPT}, 1))$ of σ'_3 is a timed trace in ATM. The reception of the tuple of values on channel Auth associates $rid\_ret, stat, mid\_ret$ respectively with $1, \text{ACCEPT}, 1$. After the prefix, the LUT produces an output on the channel Cash. The guard $\phi(tr_5^a)$ is satisfiable since $rid\_ret = 1$, $stat = \text{ACCEPT}$ and $mid\_ret = 1$. Moreover, the value transmitted on this channel is allowed according to the previous events: this value is equal to the requested amount performed on the channel Wdral at the beginning (i.e. 0). Therefore, the last event is specified by the model. According to the conformance relation tioco, the trace σ'_3 is a conforming behavior with respect to TIOSTS ATM, and the verdict *Pass* is emitted.

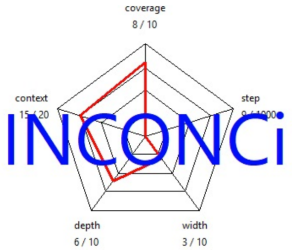



```
AVM OFF-LINE TEST  PROCESSOR
================================================================================
============================  THE VERDICT IS : PASS  ============================
================================================================================
```

$\sigma'_3 = (0, \text{Init?}).(0, \text{Wdral?}0).(0, \text{Debit!}(1,0,1)).(0, \text{Auth?}(1, \text{ACCEPT}, 1)).(0, \text{Cash!}0)$

FIGURE 3.18: Application of OFFLINE: Trace succeeds in covering test purpose

Fourth example of trace. For the last example, we take into consideration a trace of the verdict $Inc_{uspecIntI}$ in the test case $\sigma_4 = (0, \text{Init?}).(0, \text{Wdral!}0).(0, \text{Debit?}(1,0,1)).(0, \text{Auth?}(1, \text{ACCEPT}, 1)).(0, \text{Init?})$ (sequence of transitions covering $q_{ec_1}, q_{ec_2}, q_{ec_3}, q_{ec_4}$ and $Inc_{uspecIntI}$). The corresponding trace for the LUT is $\sigma'_4 = (0, \text{Init?}).(0, \text{Wdral?}0).(0, \text{Debit!}(1,0,1)).(0, \text{Auth?}(1, \text{ACCEPT}, 1)).(0, \text{Init?})$.
This trace denotes a specified behavior in the reference model, however this is not the one that we would like to observe. Indeed, after the prefix $(0, \text{Init?}).(0, \text{Wdral?}0).(0, \text{Debit!}(1,0,1)).(0, \text{Auth?}(1, \text{ACCEPT}, 1))$, the test case should receive a reaction on channel Cash in order to follow the test purpose tp_1. However, the next event is an internal reception on channel Init which is sent by another subsystem. Therefore, this reception on channel Init by the LUT leads to the verdict $Inc_{uspecIntI}$.




```
================================================================================
=========================  THE VERDICT IS : INCONCi  ===========================
================================================================================

       SYMBEX< DIVERSITY > Git-Version< ba0bfc6 > @ Oct 29 2019
       2010 - 2019 CEA List
       All Rights Reserved
       Launch @ Wed Oct 30 22:38:58 2019
```

$\sigma'_4 = (0, \text{Init?}).(0, \text{Wdral?}0).(0, \text{Debit!}(1,0,1)).(0, \text{Auth?}(1, \text{ACCEPT}, 1)).(0, \text{Init?})$

FIGURE 3.19: Application of OFFLINE:
Trace outside of test purpose due to unspecified internal input

We conclude this section by giving some metrics on some generated test cases in Table 3.1. Those are generated for the ATM model by varying the size (the length in terms of number of transitions) of the test purpose from 5 to 10, ..., 100. We find that except the generation duration, all remaining metrics increase approximately linearly with the size of the test purpose. Indeed, from a test purpose of 100 transitions, our implementation takes more than 40 seconds to generate the TIOSTS test case, compared to only 8 seconds to generate the test case for a test purpose of 50 transitions.

Size of TP	Coverage duration	Generation duration	Nb of transitions	Nb of LOC
5	40ms	500ms	31	277
10	80ms	1s160ms	62	459
15	140ms	2s110ms	87	641
25	290ms	3s220ms	138	1005
50	890ms	8s60ms	261	2778
100	3s260ms	42s150ms	513	6058

Size of TP: number of transitions in the test purpose,
Coverage duration: the duration to cover the selected test purpose,
Generation duration: the duration to generate the test case,
Nb of transitions: number of transitions in the generated test case,
Nb of LOC (Lines Of Code): number of lines of xLIA code in the test case
Experimentations on an ASUS computer, memory 8GB, processor Intel Core i7 2.00GHz $\times$ 4, Ubuntu operating system.

TABLE 3.1: Metrics on some generated test cases for ATM.

3.6 Related work

We discuss related works on Model-Based Testing approaches using the black box conformance relation ioco [78, 33] and some of its recent variants, namely tioco [46] (for timed testing) and pioco [34] (for probabilistic testing), in the unitary case.

In the decades of 90, the first MBT approach based on a conformance relation was applied for a kind of model called *Input Output Labeled Transition Systems* (IOLTS) and introduced by Tretmans [77] where each transition is labeled by an event (neither data nor time is handled). This work is considered as one of foundation approaches in formal testing based on models. The SUT's behavior is verified with respect to the model thanks to the conformance relation ioco. This relation has been then used in numerous approaches [20, 33, 18, 57].

STG [20] is a tool that has been developed to automatically derive test cases compliant with ioco, yet from a model of *Input Output Symbolic Transition Systems* (IOSTS) which can be seen as compact representations of IOLTS. It is considered as one of the first tools that have dealt with symbolic test generation. The advantage of symbolic technique is to allow the production of compact test cases with symbolic variables which need to be instantiated only during the test execution, hence enables its reuse. The process takes a specification of the system together with a test purpose which is also an IOSTS describing the behavior that one would like to test, and then produces a symbolic test case (also an IOSTS). The construction of a test purpose is manual by annotating an accepted behavior with a state *Pass* and rejected behaviors with states *Reject*. Rejected behaviors are not really erroneous, just not targeted by the test purpose. The IOSTS of the generated test case is then translated to an executable form (written in C++), which allows to perform the test execution on an SUT. The tool was then successfully used on smart card case studies. The tool is not maintained anymore.

In the approach of Krichen et al. [46], the authors have introduced an offline test generation approach for reactive systems. The specification is constructed based on the model Timed Automata (abbreviated

as TA). Those allows the definition of timed behavior in reactive systems, using clock constraints as well, yet values of data variables (other than clocks) are enumerated. They have proposed timed input-output conformance relation tioco extended from ioco. The relation is defined on the semantics of a TA which is expressed as a set of timed traces. Unlike the previously presented work [20], the test generation process takes only into consideration the reference model (without test purpose), hence the test case is directly generated from the model and is of the form TA. In this approach, the generation algorithm has been partially specified. Indeed, in cases where there are many possible stimulations from a given state in the test case or there is a non-deterministic choice between a stimulation and a delay, the choice will be made according to user-defined parameters or randomly chosen. However, the test generation is not possible in cases where the reference model containing loops defines an infinite set of possible behaviors. For this, the authors have proposed structural coverage criteria, such as state coverage or transition coverage. The approach has been implemented in a prototype tool, called TTG.

The work of Bohnenkamp et al. [11] has used TA as reference models. From such models, the authors have constructed a so-called zone-automaton which represents semantics of a TA in an abstract compact manner. This zone-automaton is a Labeled Transition System (LTS) in which states are composed of control states of the TA and a clock zone, where a zone is a symbolic denotation of all possible values that clocks can receive to satisfy the automaton guards. The idea has been to generate a test case with zones, and check on-the-fly that execution is still in a reachable zone after an action being either a input or an output. If it is the case, the test continues, otherwise the test fails. This approach has been implemented as an extension of the TorX tool [79]. We point out that the test case does not follow a test purpose, next inputs are chosen at random.

Similar to the previous works [20, 11], the authors in the work of Gaston et al. [33] have introduced an online approach for test case generation. The formalism that has been adopted to model a system is *Input Output Symbolic Transition Systems* IOSTS, which deals only with symbolic data. The particularity of this approach is that the authors have used the symbolic execution technique to unfold all possible behaviors of the reference model under the form of a symbolic execution tree. The test generation takes as input the reference model and a test purpose. However, unlike STG [20] (test purpose is manually constructed) and our approach (test purpose is a path in symbolic tree and characterized as a sequence of transitions in the model), the test purpose in this work is directly selected as a finite subtree in symbolic tree. The test generation process consists in computing stimulations allowing to follow the test purpose, and verifying whether a reaction from an SUT is specified by the reference model. To automatically define a test purpose, the approach has also proposed to use some coverage criteria such as: all symbolic behaviors of a same length or restriction by inclusion criterion (allow to extract a subtree of the symbolic execution tree by avoiding redundancies). The test generation process has been implemented in a tool called AGATHA [56].

Andrade et al. [1] have addressed the generation of online test cases from a variant of TIOSTS, yet deterministic as in our approach. While symbolic execution has been used to handle data variables, zone-based [8] abstraction techniques have been used to denote constraints on clocks. This is possible as clocks constraints are separate formulas, bounded by constants, one cannot for instance bound a clock by (the value of) another data variable. The class of TIOSTS that we have introduced are more expressive in the sense that a clock constraint is a first-order formulas, hence clocks may be constrained by other data variables, in comparison as well with the one in [5]. The test selection is different from ours: it is based on the synchronous product between the TIOSTS of the reference model and the test purpose being a TIOSTS which is defined independently. The latter represents a desired behavior (or property) of the system under test that the testing aims to check. An inconvenient of such selection is that the test purpose definition is manual and not straightforward when the model becomes complex. In our approach, a test purpose is a sub-behavior of the TIOSTS of the reference model associated with automated selection mechanism. In the authors's work, test cases are derived by transforming the symbolic execution of the synchronous of product. The test case is generated as an automaton (a TIOSTS), which enables its reuse. The implementation has not been detailed in order to

validate the generated test cases. In our approach, we offer the possibility to parameterize the generation by a set of controllable/uncontrollable inputs which enables the use of the obtained test cases in distributed testing.

In the work of Bannour et al. [5], the authors have presented an approach for applying a complete offline testing framework by using TIOSTS as reference models. This framework is based on the conformance relation tioco [46]. Authors use the notion of a test purpose which is characterized by a path in the symbolic execution of the TIOSTS. Then, an input sequence is obtained from the test purpose using constraint solving techniques to compute first a timed trace, in which then outputs are removed. The test execution consists in submitting inputs when their delays are fulfilled. Meanwhile outputs are collected together with their occurrence delays, to form an output sequence. Finally, the input sequence and output actions are merged to reconstruct a complete timed trace of SUT. A tioco-based oracle algorithm is then applied to analyze the resulting trace and compute a verdict. About the advantages of the approach, we do not waste time on solving constraints to compute a new input to stimulate the SUT. This is because input actions and delays are generated in advance. However, inconclusive verdicts can be observed more often as inputs are not computed with respect to previous observation, which can compromise the following of the test purpose.

The approach of Chimisliu et al. [18] has discussed the test case generation from UML statecharts. A reference model described in UML statechart is then encoded by LOTOS [31], a description language for formal specification. A test purpose is given as an IOLTS, and the elements for test case generation are composed of the reference model and the test purpose which represents an abstraction of the reference model describing a scenario of interest. The particularity of this approach is that the authors have allowed to annotate directly so-called refuse states in the model in order to identify parts of the reference model that will be not of interest during the generation process. This technique has allowed to select test purposes in order to limit the state space being searched. Therefore, the selection of test purpose is completely automatic by users. To evaluate the approach, they have applied to several case studies from industry by using the tool TGV [41].

In the paper of Gerhold et al. [34], the authors have presented a testing framework for probabilistic systems. Unlike other presented model, a specification is given as a Probabilistic Input Output Transition System pIOTS, which is an extension of the untimed model IOLTS where transitions from any given state are annotated by probabilities. The test case generation process has taken into account the reference model pIOTS. Like the approach [11], there is no notion of test purpose proposed by this work, test cases are generated from the reference model. The generated test cases are also probabilistic automata in which the probability is observed through the frequency when test cases are executed multiple times. The generation approach has been based on a conformance relation called pioco, an extension of ioco for probabilistic systems. The idea has consisted in constructing a set of traces in the reference model with positive probabilities and check whether a system produces a trace without belonging to the set of traces of the reference model. The approach has not developed a dedicated tool to validate the generated test case. For experimentations, the authors have then applied to two case studies by using the MBT tool JTorX [9].

The authors in the recent work [57] have proposed another approach for on-the-fly testing. In this approach, they have used the formalism IOLTS to represent specified behaviors of a reference model. The test case generation has been based on the same technique as [1, 20] when the test purpose is defined by an IOLTS other than the reference model and by annotating test purpose with accept and refuse states. Therefore test purpose construction is still manual. We point out the difference between this approach and the one presented in [18] which has proposed to annotate directly refuse states in the reference model. A test case is generated from the synchronous product of the test purpose and the reference model, and is an IOLTS armed by three verdict states *Pass*, *Fail* and *Inc*. The approach has been then implemented in the tool TESTOR and has been carried out on an important number of benchmark examples.

Approach	Time	Symbolic	Online	Relation	Tool
[77] 1996, Tretmans	×	×	×	ioco	×
[20] 2002, Duncan et al.	×	✓	✓	ioco	✓
[46] 2004, Krichen et al.	✓	×	×	tioco	✓
[11] 2005, Bohnenkamp et al.	✓	×	✓	tioco	✓
[33] 2006, Gaston et al.	×	✓	✓	ioco	✓
[1] 2011, Andrade et al.	✓	✓	✓	tioco	×
[5] 2012, Bannour et al.	✓	✓	×	tioco	✓
[18] 2013, Chimisliu et al.	×	×	✓	ioco	×
[34] 2016, Gerhold et al.	×	×	✓	pioco	✓
[57] 2018, Marsso et al.	×	×	✓	ioco	✓

TABLE 3.2: Comparison between approaches for unitary testing

3.7 Conclusion

In this chapter, we have presented two approaches in Model-Based Testing: the offline approach and the online approach. In our approach, we favor the online approach for its ability to follow the test purpose as far as possible. We have therefore defined generation of online test cases using symbolic execution techniques. The generation is based on the reference model and a test purpose characterizing a behavior that we would like to test. The generation takes into account the distinction between internal input and external input channels, which are respectively considered as uncontrollable and controllable channels from the viewpoint of a test case. This allows those test cases to be used in a distributed testing where some inputs that we call internal, are provided at runtime by other subsystems. Finally, we have implemented the generation as a module in the Diversity tool.

Chapter 4

Distributed systems: testing, coverage and derivation of scenarios

Contents

This chapter presents our contribution related to the testing framework for distributed systems, as well as its implementation in the Diversity tool. For this purpose, it will be structured as follows:

- Firstly, we present an informal description of a distributed system (denoted as DS), and its formal definition in Section 4.1;
- Next Section 4.2 introduces several problems that we often meet in DS and the need to test them to avoid erroneous implementations;
- In Section 4.3, we introduce our testing architecture;
- Then, for the purpose of testing DS, we introduce an operational semantics for DS based on symbolic execution techniques in Section 4.4;
- The implementation of our approach is presented in Section 4.5. Firstly, we discuss our approach to generate test cases for distributed systems. Secondly, we present a global test purpose which is a particular behavior in our system ATM-Bank. Additionally, to give an alternative visualization besides the traditional symbolic tree of a global test purpose in a DS, we generate it in the form of a sequence diagram, which allows to represent internal communications between subsystems and easily choose the scenario to be tested. This diagram is annotated with temporal and data constraints reflecting the feasibility of this behavior;

- Section 4.6 draws some feedback about our approach by giving its strength and limitation w.r.t other state of the art approaches;
- Finally, Section 4.7 concludes the chapter.

4.1 Distributed systems

Nowadays, distributed systems are very ubiquitous in the technology landscape. Some familiar examples of DS can be cited such as Internet, cloud of services, client-server systems, clusters of workstations, embedded systems, cyber-physical systems (IoT systems, ...) etc.

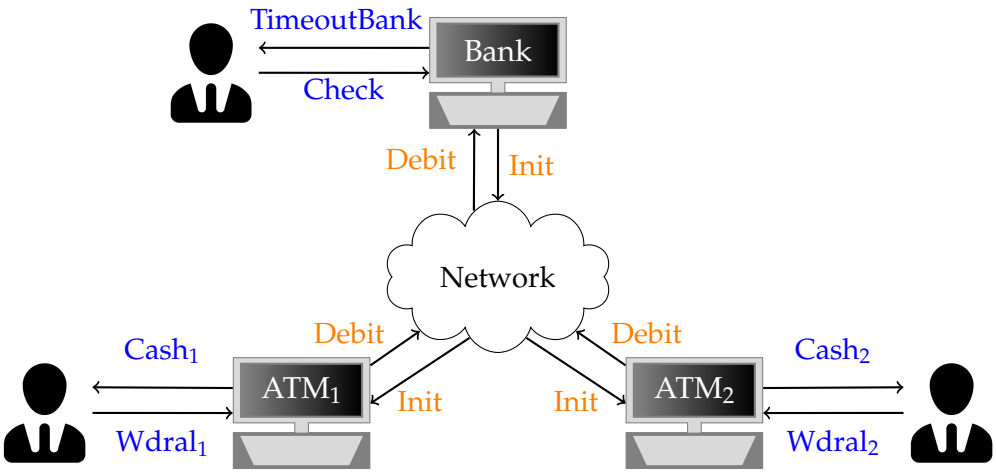


FIGURE 4.1: A sample of DS: The distributed system ATM-Bank

Distributed systems are composed of intercommunicating localized subsystems deployed on distant machines. These subsystems are interconnected by a communication networks, and exchange messages to achieve the system goals. In Figure 4.1, our distributed system ATM-Bank is composed of a bank denoted by Bank and many machines ATM deployed in different cities communicating through communication channels like Check, Init, Wdral, ... We illustrate only two ATM and several communication channels in Figure 4.1. And in the following, for the sake of simplicity, only one ATM and one Bank are considered without loss of generality.

In our approach, a distributed system is defined as an n-tuple of subsystems. And the model for a DS is a tuple $Sys = (\mathbb{G}_1, \ldots, \mathbb{G}_n)$ where $\mathbb{G}_i$ $(1 \leq i \leq n)$ is a TIOSTS with signature $\Sigma_i = (\Omega, A_i, K_i, C_i)$ (cf. Subsection 2.2.1). For all subsystems $\mathbb{G}_i$ and $\mathbb{G}_j$, $i \neq j$, we have $A_i \cap A_j = \emptyset, K_i \cap K_j = \emptyset$, i.e. the sets of variables (resp. the sets of clocks) of any two subsystems are disjoint. The subsystems interact through a *distributed interface*, which is defined as an n-tuple of set of channels of subsystems, i.e. $\Lambda = (C_1, \ldots, C_n)$. In our work, we allow multicast communications, i.e. a subsystem can send an output to many other subsystems which receive this input on a same channel. With $\Lambda = (C_1, \ldots, C_n)$ the distributed interface of Sys, we adopt the notations:

- $C(\Lambda) = \bigcup_{i \leq n} C_i$ is the set of all channels in the distributed system Sys. $C(\Lambda)^{in} = \bigcup_{i \leq n} C_i^{in}$ is the set of all input channels and $C(\Lambda)^{out} = \bigcup_{i \leq n} C_i^{out}$ is the set of all output channels in Sys.

- $C(\Lambda)^{int} = \bigcup_{i \leq n} C_i^{int}$ where $C_i^{int} = \{C_i \cap C_j \mid 1 \leq j \leq n \wedge j \neq i\}$ denotes the set of internal channels of the subsystem $\mathbb{G}_i$
- $C(\Lambda)^{ext} = \bigcup_{i \leq n} C_i^{ext}$ where $C_i^{ext} = C_i \setminus C_i^{int}$ denotes the set of external channels of the subsystem $\mathbb{G}_i$

which implies that for all subsystems $\mathbb{G}_i$ and $\mathbb{G}_j$, $i \neq j$, we have $C_i^{ext} \cap C_j^{ext} \neq \emptyset$.

Example 4.1.1. In this example, we consider the following distributed system $Sys_{AB} = (\text{ATM}, \text{Bank})$ which is composed of one ATM machine and one Bank. Let take a look of the communication channels between the ATM and the Bank as well as between these subsystems with the local environment in Figure 4.2.

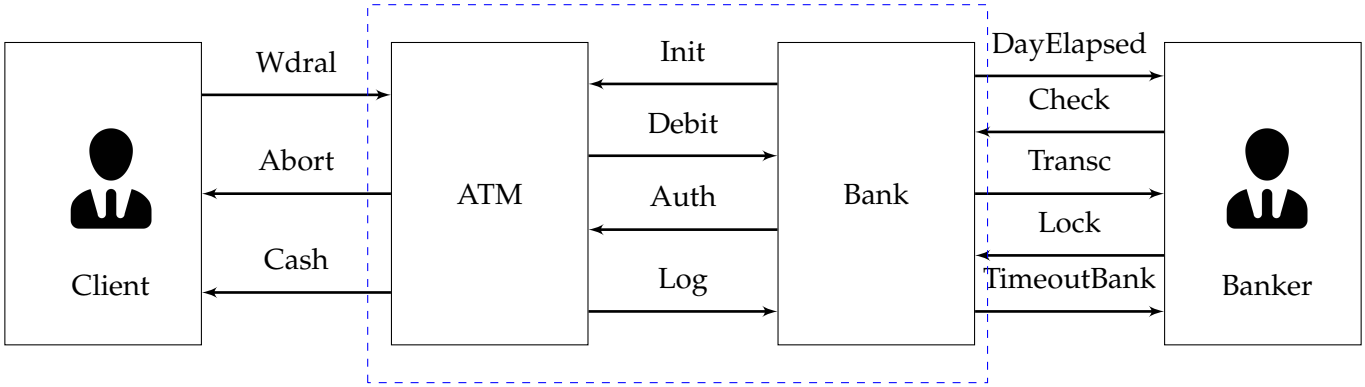


FIGURE 4.2: Communication channels of $Sys_{AB} = (\text{ATM}, \text{Bank})$ with environment

The signature of ATM and Bank is respectively defined as (Ω, A_a, K_a, C_a) and (Ω, A_b, K_b, C_b) in the table 4.1.

		ATM	**Bank**
Variables		$A_a = \{rid, amt, fee, rid\_ret, stat, mid\_ret\}$	$A_b = \{days, sum, cpt, tloc, tid, x, aid, bal\}$
Clocks		$K_a = \{wclock\}$	$K_b = \{tclock, dclock, pclock\}$
Channels	**Input**	$C_a^{in} = \{\text{Wdral}, \text{Init}, \text{Auth}\}$	$C_b^{in} = \{\text{Debit}, \text{Log}, \text{Check}, \text{Lock}\}$
	Output	$C_a^{out} = \{\text{Abort}, \text{Cash}, \text{Debit}, \text{Log}\}$	$C_b^{out} = \{\text{Init}, \text{Auth}, \text{DayElapsed}, \text{Transc}, \text{TimeoutBank}\}$
	Internal	$C_a^{int} = \{\text{Init}, \text{Debit}, \text{Auth}, \text{Log}\}$	$C_b^{int} = \{\text{Init}, \text{Debit}, \text{Auth}, \text{Log}\}$
	External	$C_a^{ext} = \{\text{Wdral}, \text{Abort}, \text{Cash}\}$	$C_b^{ext} = \{\text{DayElapsed}, \text{Check}, \text{Transc}, \text{Lock}, \text{TimeoutBank}\}$

TABLE 4.1: Signature of subsystems and distributed interface in Sys_{AB}

We recall in the following the functionality of our system ATM-Bank: the TIOSTS ATM (Figure 2.4) is used to receive withdrawal requests from clients stored in the variable *amt* on channel Wdral and transfer

requests to the Bank on channel Init and waiting for a reply on Auth. A transferred request is composed of the request identifier (rid), the requested amount with the charge ($amt + fee$) and the machine identifier (ATM_ID). After receiving a reply to a request on channel Auth, the ATM returns cash on channel Cash if the client is allowed to withdraw this sum, otherwise the request is aborted on channel Abort.

The TIOSTS Bank (Figure 2.5) uses a circular array of size 3 (i.e. sum) to store withdrawal amounts of the last three days [1]. The counter $days$ is incremented at the end of each day and is used to identify the next place in the array sum to store the accumulated amounts for the new day by using the operator modulo. Bank receives the withdrawal request on channel Debit from ATM. The received data is stored in the tuple (tid, x, aid). Then Bank assigns authorizations to this request on channel Auth by emitting one among three responses: Accept, Reject or Lock. In order to decide to emit which response, the Bank verifies if the requested amount does not exceed the balance bal and the total debit limit over a rolling 3 days is not exceeded. The Bank allows also a banker to send a request on channel Check, and receive a reply which is the number of successful requests within a day on channel Transc, then to modify the lock time on channel Lock if he finds that there are too many transactions during some period.

In addition to traditional operations such as withdrawal request, balance verification for withdrawing, and cash return, our system ATM-Bank allows also security processes, e.g. in particular the verification of consecutive withdrawals within some too short duration protects against the situation where someone steals a card and tries to withdraw as much money as possible.

Notation 4.1.1. For a subsystem $\mathbb{G}_i$, we recall that $UC^{in}(\mathbb{G}_i)$ denotes the set of uncontrollable input channels of $\mathbb{G}_i$, i.e. $UC^{in}(\mathbb{G}_i) = C_i^{in} \cap C_i^{int}$ and $CC^{in}(\mathbb{G}_i)$ the set of controllable input channels of $\mathbb{G}_i$, i.e. $CC^{in}(\mathbb{G}_i) = C_i^{in} \cap C_i^{ext}$ (cf. Section 3.2.1). This allows us to differentiate between those channels ($UC^{in}(\mathbb{G}_i)$) used by $\mathbb{G}_i$ to receive inputs from other subsystems of Sys and those ($CC^{in}(\mathbb{G}_i)$) used to receive stimulus from the external environment.

Example 4.1.2. The set of uncontrollable input channels of ATM and Bank is respectively defined as follows:

$UC^{in}(\text{ATM}) = \{\text{Init}, \text{Auth}\}$

$UC^{in}(\text{Bank}) = \{\text{Debit}, \text{Log}\}$

The set of controllable input channels of ATM and Bank is respectively defined as follows:

$CC^{in}(\text{ATM}) = \{\text{Wdral}\}$

$CC^{in}(\text{Bank}) = \{\text{Check}, \text{Lock}\}$

We have just introduced in this section the informal description of a distributed system, as well as its formal definition and some notations about its distributed interface. In the following, we will talk about the problems that we often encounter while constructing distributed systems and therefore, the need for testing them to avoid incorrect operations.

4.2 The interest of testing distributed systems

In spite of being very widely used, the implementation of a distributed system can make well-known failures. Certain failures can be classified at a local level such as:

- Timing failure - a response time to a request is not respected, e.g. a subsystem takes too long (more than the specified interval) to reply to a request
- Response failure - the response to a request is incorrect, e.g. a subsystem provides an unexpected response on an output channel

[1] For illustrative purposes, we choose an interval of three days instead of seven days, which is common for most of banks.

Some other failures at the global level are the ones in the internal messages between subsystems:

- Corrupted message occurs when an internal communication is injected by a hacker, i.e. he supplies internal input to a subsystem, which produces a security risk: the communication that is not emitted by a subsystem is finally received by another subsystem in the distributed system.
- Lost messages is typically the consequence of internal communication failures. Indeed, a message can be lost as someone tries to catch or hide it to prevent the distributed system from correctly operating

Because of these typical failures, testing DS to detect such kinds of failures before operating has an extremely important role. However, this process is recognized to be difficult due to a number of characteristics of distributed systems, including:

- non-deterministic situation: the absence of a global clock makes distributed systems become non-deterministic [55]. Without a global clock, subsystems cannot synchronize their local behaviors. For example, a reception for a subsystem can happen before or after another one, leading to two different situations, which gives two behaviors to be taken into consideration;
- lack of reliable communications media: this leads to the problem of loss of messages (a) or latency (b), etc. For example, in case (a), a message which is lost during its transmission due to a communication error can cause an undesired behavior where the subsystem that needs to receive the message is always waiting for it to start a process. In case (b), a big latency in the transmission of a request can delay the reception of this request at a subsystem, inducing an incorrect operation of the system.

For the purpose of illustrating the non-deterministic situations, let consider two diagrams in Figure 4.3 which describe the interactions of the system Sys_{AB}: the reception of the message on channel Check by the subsystem Bank can arrive before or after the reception of the message on channel Debit, which gives two different traces for the subsystem Bank:

$\sigma_1 = (1, \text{Init!}).(3, \text{Debit?}(1, 10, 1)).(2, \text{Auth!}(1, \text{ACCEPT}, 1)).(1, \text{Check?}).(0.5, \text{Transc!2}).(0.5, \text{Lock?10})$ in the diagram on the left: The banker sends a lock request to lock transitions during some period (10 time units) after the withdrawal request reception, therefore the request is accepted.

$\sigma_2 = (1, \text{Init?}).(1, \text{Check?}).(0.5, \text{Transc!2}).(0.5, \text{Lock?10}).(1, \text{Debit?}(1, 10, 1)).(2, \text{Auth!}(1, \text{LOCK}, 1))$ in the diagram on the right: The banker sends a lock request at the beginning, then the withdrawal request is locked and there is no cash returned to the client.

Indeed, the user interacting locally with Bank cannot control the moment of emission of the message on channel Check since he does not know exactly when he has to emit it. These two situations represent two different traces. We can see that the interleaving of traces due to uncontrollable channels introduces more and more behaviors to manage, making distributed systems difficult to design and build.

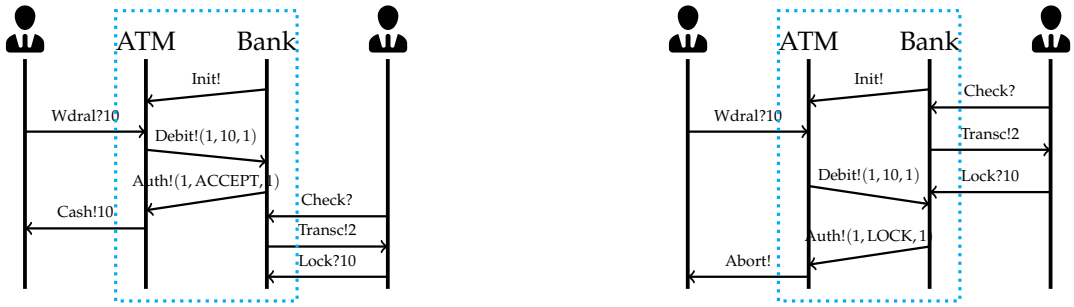


FIGURE 4.3: Non-deterministic situations

Because of these major obstacles, testing and analyzing behaviors of a distributed system, in particular those describing scenarios involving many subsystems, faces many difficulties. Each kind of failure that we have mentioned could be an interesting scenario for testing if we can define it. This scenario is called *global test purpose*, which is a necessary ingredient to our testing architecture that will be presented in the following.

4.3 Our testing architecture for distributed systems

As explained in Chapter 3, when it comes to testing a Localized System Under Test (LUT), we consider specific behaviors that are characterized by a *test purpose* relevant for testing, for which we define test cases aiming at stimulating the LUT, pursuing the goal of covering those behaviors.

In this section, we present our testing architecture and different hypothesis. When testing DS, we need to interact with distant subsystems [82], either by a global test case which interacts with all subsystems or by a collection of local test cases where each of them interacts with its localized subsystem. Our architecture (Figure 4.4) includes both previous configurations when it is composed of local, autonomous test cases, each of which interacts with its localized subsystem. Moreover, we have a global tester whose role is to emit two coordination messages to all local test cases: one at the beginning to initialize and one at the end to terminate the testing process. In the literature, local test cases can exchange coordination messages or not [82]: 1) local test cases regularly synchronize their actions by sending coordination messages to inform other local test cases of their progress in the global testing process 2) a local test case interacts only with its subsystem and do not know about the evolution of other distant local test cases. In the first case, coordination messages assure a consistent global view of the system execution, but require some other hypotheses due to the delay of coordination messages [83]. Our testing architecture is constructed based on the second one.

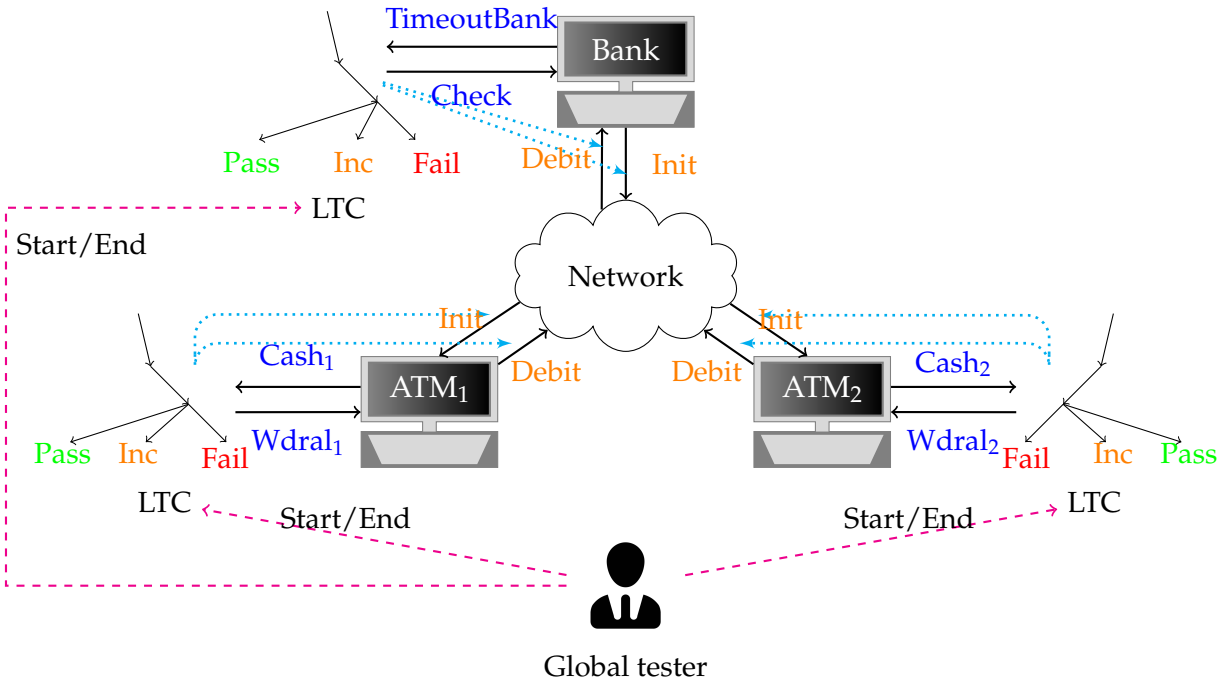


FIGURE 4.4: Our testing architecture

In our testing architecture, each local test case (denoted as LTC) is placed and is in direct interaction with its localized subsystem. We do not have the coordination between test cases. The proposed architecture induces the controllability problem by considering that local test cases can observe internal messages received or sent by the localized subsystems. Such hypotheses require more testing instrumentation efforts than simple test cases playing the role of the environment. However, they increase the number of observations on the system which, based on the causal relation between emission and reception of any internal message, facilitate the scheduling of local events to reconstruct a consistent global scenario. Unlike the architecture in the work [10] while one collect local traces for a posterior verification, local test case benefits from constraint solving techniques to compute next inputs based on previous outputs, allowing to pilot the test purpose.

We have seen in Chapter 2 (see Section 2.4.3), that symbolic execution is a useful technique for characterizing precisely test purposes as being a sequence of consecutive transitions of a reference TIOSTS $\mathbb{G}$ since it allows to compute their counterpart in the form of satisfiable symbolic paths of $SE(\mathbb{G})$. Moreover, symbolic execution generates useful pieces of information resulted from internal communications between subsystems which are interesting for the test case generation process. For this reason, this technique will be discussed in the following section.

4.4 Symbolic execution for testing distributed systems

In order to obtain behaviors of a distributed system, we introduce in this section an operational semantics allowing to execute symbolically the distributed system. Then, for illustrative purposes, we apply the symbolic execution to our system ATM-Bank.

4.4.1 Distributed system context

The symbolic behavior of a DS model is defined on top of that for TIOSTS by adding means that handle internal communications and reason about their causality. This mainly comes to store internal receptions and ensure that each of them corresponds to a previous emission which allows us to rule out anomalies such, for example, that the reception of an internal message by a subsystem is allowed only if the message has been previously sent by another subsystem. To capture the information necessary to deal with such anomalies, we introduce the notion of distributed execution context for keeping track of additional information so that the reception of an internal message by a TIOSTS is allowed only if the message has been previously sent by another TIOSTS.

In Section 2.3 of Chapter 2, we introduced the symbolic execution of a unitary system $\mathbb{G}$. Its symbolic execution is processed based on execution contexts which contain elements related to the execution. The set of execution contexts is denoted as $\mathbb{EC}$. For the distributed system $Sys = (\mathbb{G}_1, \ldots, \mathbb{G}_n)$, apart from the execution context for each subsystem, we need other pieces of information related to the execution of internal communications. We extended the notion of execution context for distributive purpose. The new extended notion called *distributed execution context* of the subsystem $\mathbb{G}_i$ (defined over the signature $\Sigma_i = (\Omega, A_i, K_i, C_i)$) is defined as a tuple $dec = (ec, \gamma, \chi, d, tr\text{-}seq)$ where

- $ec \in \mathbb{EC}$ is the reached execution context of $\mathbb{G}_i$ being executed in the context of Sys;
- $\gamma : UC(\mathbb{G}_i) \rightarrow (\mathcal{T}_\Omega(F_d) \times \mathcal{T}_\Omega(F_t))^*$ associates each uncontrollable channel with its pending content in terms of a received piece of data and its emission date where F_t and F_d are respectively the set of fresh variables to denote variables of type time and variables of other types (cf. Section 2.3). $\mathcal{T}_\Omega(F_d)$ and $\mathcal{T}_\Omega(F_t)$ are respectively the set of terms constructed over F_d and F_t. We note that the pending content of each uncontrollable channel is considered as an unbounded fifo-queue, i.e. the first data received is the first one to be consumed;

- $\chi \in \mathcal{F}_\Omega(F)$ represents a constraint on time and data inferred from data exchanges between $\mathbb{G}_i$ and other subsystems in the context of Sys;
- $d \in \mathcal{T}_\Omega(F_t)$ is the date at which ec has been reached;
- $tr\text{-}seq \in Tr_i^*$ is the sequence of transitions of $\mathbb{G}_i$ that has been executed leading to ec from initialization.

Notation 4.4.1. For any $dec = (ec, \gamma, \chi, d, tr\text{-}seq)$, $ec(dec)$, $\gamma(dec)$, $\chi(dec)$, $d(dec)$ and $tr\text{-}seq(dec)$ stand for respectively ec, γ, χ, d and $tr\text{-}seq$.

The Distributed System Context (abbreviated by DSC) of a DS model $Sys = (\mathbb{G}_1, \ldots, \mathbb{G}_n)$ is a tuple $dsc = (dec_1, \ldots, dec_n, psc)$ where psc is the predecessor DSC from which dsc has been created by executing the corresponding transition. As for initial execution context, this component is left undefined for an initial system execution context.

Notation 4.4.2. For any $dsc = (dec_1, \ldots, dec_n, pdsc)$, $pdsc(dsc)$ stands for $pdsc$.

The symbolic execution of a DS model consists of executing a transition tr of its subsystems $\mathbb{G}_i$ and making the DS distributed execution context evolves accordingly. Intuitively, the evolution of the distributed system context will essentially concern the distributed execution contexts of only subsystems concerned by the transition tr, i.e. whose signature contains the channel of the transition $chan(tr)$ (cf. notation 2.2.1).

4.4.2 Symbolic execution rules for distributed systems

Notation 4.4.3. In order to take advantage that only some elements of DSC are likely to be modified at each symbolic execution step, we will use the notation $[\![\ ]\!]$ which will highlight the only modified components. For example, with $dsc' = (dec'_1, \ldots, dec'_n, psc)$, $dsc' = dsc[\![d(dec'_i) \leftarrow d(dec_i) + delay]\!]$ means that dsc' coincides with dsc except that the component d of the i-th distributed execution context is increased of the value $delay$.

The symbolic execution of a transition tr of the subsystem $\mathbb{G}_i$ from $dsc = (dec_1, \ldots, dec_n, psc)$ with $src(tr) = q(ec(dec_i))$ allows to compute a successor system context $dsc' = (dec'_1, \ldots, dec'_n, dsc)$ of dsc by considering a temporary system context $dsc^{tmp} = (dec_1^{tmp}, \ldots, dec_n^{tmp}, dsc)$ defined by:

$$dsc^{tmp} = dsc \left[\!\!\left[\begin{array}{l} ec(dec_i^{tmp}) \leftarrow ec', \\ d(dec_i^{tmp}) \leftarrow d(dec_i) + delay(ec'), \\ tr\text{-}seq(dec_i^{tmp}) \leftarrow tr\text{-}seq(dec_i).tr \end{array} \right]\!\!\right]$$

where ec' is the execution context resulted from $ec(dec_i)$ by executing tr.

- if $chan(ec') \in C(\Lambda)^{ext}$, i.e. transition tr is an external transition [2], then we have $dsc' = dsc^{tmp}$;
- if $chan(ec') = c$ and $c \in C(\Lambda)^{out} \cap C(\Lambda)^{int}$ and $act(ec') = c!t$, i.e. transition tr is an internal output transition, then for any subsystem $\mathbb{G}_j$ receiving this output, its waiting queue $\gamma(dec'_j)(c)$ is accumulated with $(t, d(dec_i^{tmp}))$. Therefore, new distributed system context dsc' is:

$$dsc' = dsc^{tmp} \left[\!\!\left[\gamma(dec'_j)(c) \leftarrow \gamma(dec_j^{tmp})(c).(t, d(dec_i^{tmp})) \right]\!\!\right]$$

with $j \neq i, j \leq n, c \in UC^{in}(\mathbb{G}_j)$

[2] An external transition is the one whose action is an external communication. An internal transition is the one whose action is an internal communication.

- if $chan(ec') = c$ and $c \in C(\Lambda)^{in} \cap C(\Lambda)^{int}$ and $act(ec') = c?x$, i.e. transition tr is an internal input transition, if the waiting queue $\gamma(dec_i)(c)$ is of the form $(term, delay).w$, then the new distributed system context dsc' is:

$$dsc' = dsc^{tmp} \left[\!\!\left[\begin{array}{l} \gamma(dec_i')(c) \leftarrow w \\ \chi(dec_i') \leftarrow \chi(dec_i^{tmp}) \wedge (d(dec_i^{tmp}) \geq delay) \wedge (x = term) \end{array} \right]\!\!\right]$$

  Otherwise if the waiting queue is empty, i.e. $\gamma(dec_i)(c) = \epsilon$, the transition tr cannot be executed.

Condition $\chi(dec_i') \leftarrow \chi(dec_i^{tmp}) \wedge (d(dec_i^{tmp}) \geq delay) \wedge (x = term)$ ensures that $term$ may be consumed only if its emission date ($delay$) is anterior to the its reception date on x ($d(dec_i^{tmp})$) and $term$ is identified by a local variable x of the subsystem receiving the message.

Remark 4.4.1. In our approach, we suppose that all subsystems $\mathbb{G}_1, \ldots, \mathbb{G}_1$ are initialized from an assumed common instant of time and the time elapsed from this moment to their boot are respectively denoted as $z_0^1, \ldots, z_0^n$.

Notation 4.4.4. In the following, we denote by $ev(dsc')$ the symbolic event of the execution context ec', i.e. $ev(dsc') = ev(ec')$. Similarly, $act(dsc')$, $delay(dsc')$ and $chan(dsc')$ represents respectively $act(ec')$, $delay(ec')$ and $chan(ec')$.

The symbolic execution $SE(Sys)$ of Sys denoted by $(dsc_1, \mathbb{DSC})$ where:

- dsc_1 is an arbitrary initial DSC $(dec_1^1, \ldots, dec_n^1, self)$ which is defined by
  - for $i \leq n$, $dec_i^1 = (ec_1^i, \gamma_1^i, true, z_0^i, \epsilon)$ with z_0^i a fresh variable
  - γ_1^i verifies that for $c \in UC^{in}(\mathbb{G}_i)$, $\gamma_1^i(c) = \epsilon$
  - $self$ indicates that the initial distributed system context has no predecessor context, except by convention the initial context itself
- $\mathbb{DSC}$ is the set of all distributed system contexts in the symbolic execution of Sys.

A symbolic system path p_{sys} of a DS is either the empty sequence ϵ or a sequence $dsc_1 \ldots dsc_m$ such that for all $i < m$, $pdsc(dsc_{i+1}) = dsc_i$. For a non-empty path p_{sys}, we denote $tgt(p_{sys}) = dsc_m$.

4.4.3 Illustration with ATM-Bank

In this section, we illustrate an example of symbolic execution in the distributed system Sys_{AB} from example 4.1.1. In the following, the subsystem ATM is abbreviated as a and the subsystem Bank as b and their sets of execution contexts are respectively denoted as $\mathbb{EC}_a$ and $\mathbb{EC}_b$. We recall that the TIOSTS ATM and Bank are respectively illustrated in Figure 2.4 and 2.5.

Initialization: Let us consider the following initial system context $dsc_1 = (dec_a^1, dec_b^1, self)$ where $dec_a^1 = (ec_1^a, \gamma_1^a, true, z_0^a, \epsilon)$ is the distributed execution context of the ATM with:

- $ec_1^a = (a_0, true, \{rid := rid_0, amt := amt_0, fee := fee_0, wclock := 0\}, (0, \epsilon), self) \in \mathbb{EC}_a$ is the initial context of ATM;

- γ_1^a associates each uncontrollable input channel of ATM with its pending content: $\gamma_1^a(\text{Init}) = \epsilon$ and $\gamma_1^a(\text{Auth}) = \epsilon$. We note that ϵ means the empty content.

and $dec_b^1 = (ec_1^b, \gamma_1^b, true, z_0^b, \epsilon)$ the distributed execution context of the bank agency with:

- $ec_1^b = (b_0, true, \{days := days_0, sum := \{sum[0]_0, sum[0]_1, sum[0]_2\}, cpt := cpt_0, tloc := tloc_0, tclock := 0, dclock := 0, pclock := 0\}, (0, \epsilon), self) \in \mathbb{EC}_b$ is the initial context of the bank agency;
- $\gamma_1^b(\text{Debit}) = \epsilon$ and $\gamma_1^b(\text{Log}) = \epsilon$.

We find that the time elapsed from the assumed common instant to the boot of the subsystem ATM and Bank are respectively denoted as z_0^a and z_0^b.

Let suppose that the distributed system has already executed the transitions tr_1^b of Bank and tr_1^a, tr_2^a of ATM as presented in Figure 4.5. The execution of these transitions leads respectively to the creation of dsc_2, dsc_3 and dsc_4.

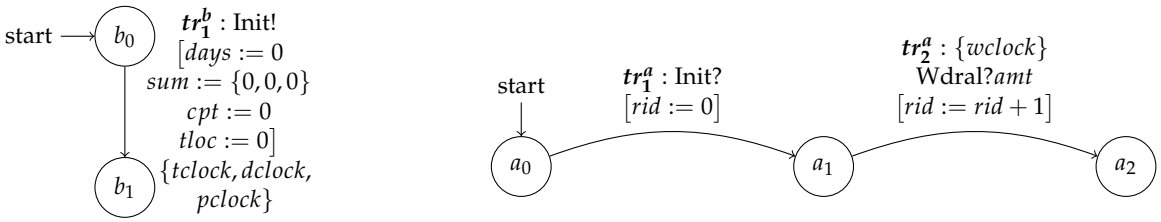


FIGURE 4.5: Transitions of ATM and Bank that have been executed

Execution of an internal output transition: The execution of transition $tr_3^a : a_2 \rightarrow a_3$ (Figure 4.6) creates the new system context $dsc_5 = (dec_a^5, dec_b^5, dsc_4)$ from dsc_4. The distributed execution context of the ATM and the Bank are respectively denoted as $dec_a^5 = (ec_5^a, \gamma_5^a, \chi_5^a, d_5^a, tr\text{-}seq_5^a)$ and $dec_b^2 = (ec_5^b, \gamma_5^b, \chi_5^b, d_5^b, tr\text{-}seq_5^b)$. The determination of the elements in the context ec_5^a is straightforward based on the principle of symbolic execution (Section 2.3.1):

Element	**Value**
$q(ec_5^a)$	a_3
$\pi(ec_5^a)$	$z_3^a \leq 1$
$\lambda(ec_5^a)$	$rid := 1,\ amt := amt_1,\ fee := fee_0,\ rid\_ret := rid\_ret_0,$ $stat := stat_0,\ mid\_ret := mid\_ret_0,\ wclock := z_3^a$
$ev(ec_5^a)$	$(z_3^a, \text{Debit!}(1, amt_1 + fee_0, ATM\_ID))$
$pec(ec_5^a)$	ec_4^a

TABLE 4.2: Different elements of ec_5^a

The new system context $dsc_5 = (dec_a^5, dec_b^5, dsc_4)$ and its predecessor dsc_4 differ only in the following elements:

- the new context of ATM is updated with ec_5^a;
- the creation date of ec_5^a is accumulated with $delay(ec_5^a) = z_3^a$, i.e. its creation date is $z_0^a + z_1^a + z_2^a + z_3^a$;
- the sequence of covered transitions is currently $tr\text{-}seq(dec_4^a) \cdot tr_3^a$, i.e. $tr_1^a \cdot tr_2^a \cdot tr_3^a$;
- the content of the input channel Debit for Bank is updated with $((1, amt_1 + fee_0, ATM\_ID), z_0^a + z_1^a + z_2^a + z_3^a)$ where $(1, amt_1 + fee_0, ATM\_ID)$ denotes the tuple of data transmitted on channel Debit: $\gamma(dec_b^5)(\text{Debit}) = ((1, amt_1 + fee_0, ATM\_ID), z_0^a + z_1^a + z_2^a + z_3^a)$.

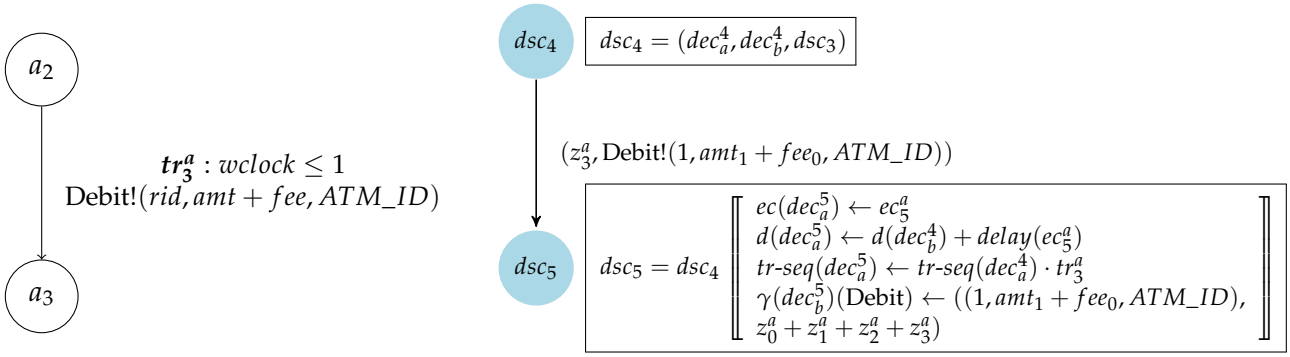


FIGURE 4.6: Symbolic execution of tr_3^a

Execution of an internal input transition: Now, let consider the symbolic execution of the reception on the channel Debit of the Bank (transition tr_2^b) which creates the new system context $dsc_6 = (dec_a^6, dec_b^6, dsc_5)$ (Figure 4.7). The distributed execution context of the ATM and the Bank is denoted as $dec_a^6 = (ec_6^a, \gamma_6^a, \chi_6^a, d_6^a, tr\text{-}seq_6^a)$ and $dec_b^6 = (ec_6^b, \gamma_6^b, \chi_6^b, d_6^b, tr\text{-}seq_6^b)$. The determination of elements of new context ec_6^b of the Bank is defined as follows:

Element	**Value**
$q(ec_6^b)$	b_2
$\pi(ec_6^b)$	$z_2^b < 86394$, determined from the constraint $dclock + 6 < 24\text{H}$ with $24\text{H} = 86400$ time units
$\lambda(ec_6^b)$	$tclock := z_2^b,\ dclock := z_2^b$
$ev(ec_6^b)$	$(z_2^b, \text{Debit?}(tid_1, x_1, aid_1))$
$pec(ec_6^b)$	ec_5^b

TABLE 4.3: Different elements of ec_6^b

To resume, the new system context $dsc_6 = (dec_a^6, dec_b^6, dsc_5)$ and its predecessor dsc_5 differ in the following components:

- the new context of Bank is updated with ec_6^b;
- the creation date of ec_3^b is accumulated with $delay(ec_6^b)$ (i.e. $z_0^b + z_1^b + z_2^b$);

- the sequence of covered transitions of the bank is currently $tr_1^b \cdot tr_2^b$;
- the only element in the pending content associating to the uncontrollable input channel Debit of the Bank is removed (i.e. $((1, amt_1 + fee_0, 1), z_0^a + z_1^a + z_2^a + z_3^a)$), because of the reception on this channel. Consequently, $\gamma(dec_b^6)(\text{Debit}) = \epsilon$;
- $\chi(dec_6^b) = (z_0^b + z_1^b + z_2^b \geq z_0^a + z_1^a + z_2^a + z_3^a) \wedge (tid_1 = 1 \wedge x_1 = amt_1 + fee_0 \wedge aid_1 = 1)$.

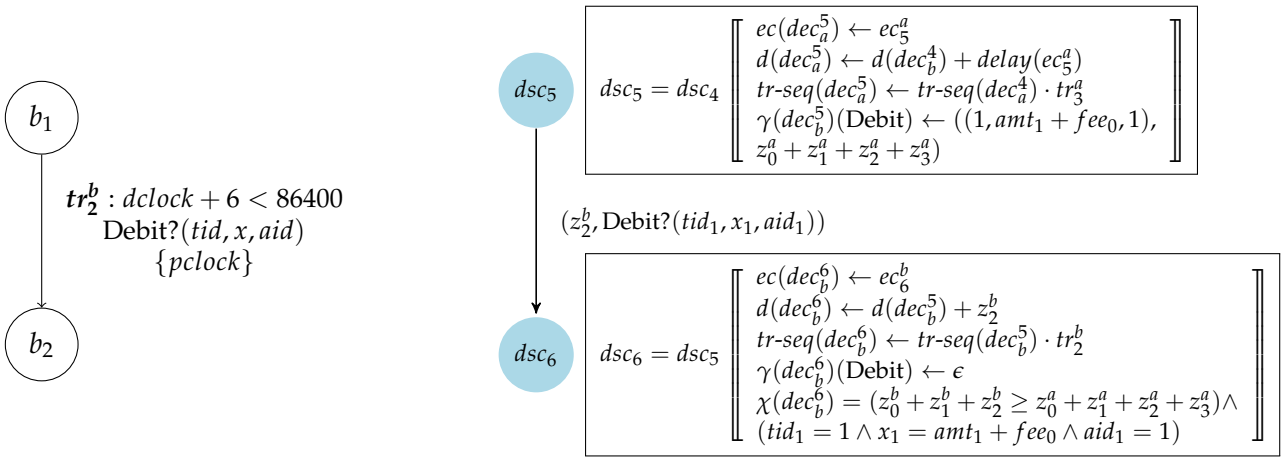


FIGURE 4.7: Symbolic execution of tr_2^b

The beginning of a test purpose for our system ATM-Bank: In the following, we will discuss the testing issues that are required by the symbolic execution of a DS. In other words, the definition of symbolic execution for a distributed system that we have previously introduced helps to show out these issues. Figure 4.8 presents a simple scenario which is generated from the symbolic execution of DS Sys_{AB}.

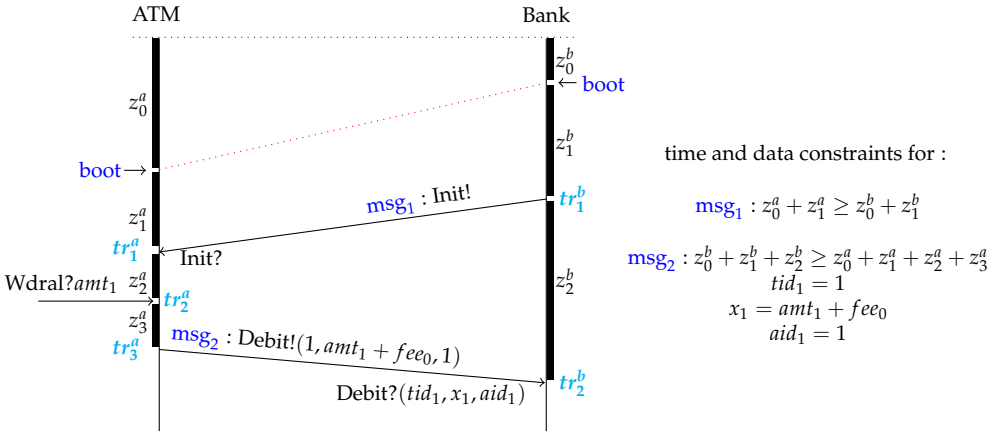


FIGURE 4.8: Illustration of the beginning of a test purpose for system Sys_{AB}

The scenario shows that the symbolic execution of the overall model introduces for both subsystems ATM and bank, an initial duration which is respectively z_0^a and z_0^b. These durations denote the time elapsed from an assumed common instant of time (depicted by a dotted blue horizontal line in the scenario depiction) to the boot of either subsystems (short arrows in the scenario). Instants z_0 and z_1 allow to establish a precedence

constraint between the date of emission of a piece of data (on an internal channel) and its later reception by another subsystem called *causality relation*, i.e. a message has to be received after its emission. Typically, the following constraint $z_0^a + z_1^a \geq z_0^b + z_1^b$ reflects that the emission of the first message msg_1 (Init! at $z_0^b + z_1^b$) by the bank agency happens before its corresponding reception (Init? at $z_0^a + z_1^a$) by the ATM machine since the reference time instant. Similarly, the same reasoning should be applied for the second message msg_2 in order to obtain the constraint $z_0^b + z_1^b + z_2^b \geq z_0^a + z_1^a + z_2^a + z_3^a$. Besides, the data identification constraints $tid_1 = 1, x_1 = amt_1 + fee_0, aid_1 = 1$ take into account that data received by the bank necessarily correspond to data emitted by the ATM (on that same channel). The variables tid_1, x_1, aid_1 are new fresh variables created by the bank in order to store values received on the internal channel Debit.

We point out that temporal correlations imply that since a duration between an emission of a message msg_1 followed by a reception of a message msg_2 at a given location (bank) should be greater than the duration between the reception of msg_1 followed by the emission of msg_2 at a remote location (ATM). All such correlation constraints are checked to be compatible with local path conditions discussed previously, in which case the scenario is a possible distributed behavior of the overall model.

4.5 Our testing framework for distributed systems

Inspired by the coverage of a sequence of consecutive transitions in Subsection 2.4.3, our implementation of testing framework for distributed systems takes into consideration a sequence of consecutive transitions involving two subsystems of Sys_{AB}, in a similar way to unitary systems. Given this sequence of transitions, the idea is to pilot the symbolic exploration with the objective to compute feasible system paths covering the sequence. In this section, firstly, we introduce our testing framework to generate test cases for distributed systems which is also based on the notion of global test purpose. Then, we illustrate a testing scenario through an example of global test purpose.

4.5.1 Our testing framework

As we know, symbolic execution computes executions of the DS model, which facilitates model's understanding, especially in case of distributed systems. Therefore, it is very practical to obtain a *global test purpose* defined by one of such execution paths which covers an interesting behavior of the system. Concretely, a global test purpose can be seen as interactions between different subsystems. In practice, these interactions are characterized by a tuple of sequence of consecutive transitions. Indeed, for the global test purpose p_{sys} represented by the sequence $dsc_1 \ldots dsc_m$ with $dsc_m = (dec_1, \ldots, dec_n, dsc_{m-1})$, the tuple of sequence of transitions $(tr\text{-}seq(dec_1), \ldots, tr\text{-}seq(dec_n))$ is used to characterize the *local test purposes*, where each component in the tuple is a sequence of consecutive transitions of one the constituent TIOSTS $\mathbb{G}_i$ with $1 \leq i \leq n$. Thus, each component in this tuple allows to obtain a *local test purpose* for each subsystem (considered as LUT) of the DS.

In agreement with models of DS given in Section 4.1, a Distributed System Under Test (DUT) is seen as a tuple of Localized System Under Test (LUT): DUT $= (\text{LUT}_1, \ldots, \text{LUT}_n)$ where each LUT_i is a subsystem. Let us introduce the main steps of our testing framework for DS as follows:

1. Selection of a global test purpose p_{sys} as a path $dsc_1 \ldots dsc_m$ in a symbolic execution tree of a distributed system;
2. Canonical derivation of the tuple $(tr\text{-}seq(dec_1), \ldots, tr\text{-}seq(dec_n))$ from p_{sys} to obtain a tuple of local test purposes $(p_{\mathbb{G}_1}, \ldots, p_{\mathbb{G}_n})$;
3. Generation of n local test cases according to n local test purposes as presented in the process in Section 3.3;

4. Initialization phase by sending to local test cases the start signal of local test executions;
5. Concurrent execution of local test cases with the goal to follow their local test purposes, without sending coordination messages toward some other test cases, but observing internal messages exchanged between LUTs;
6. Computation of a global verdict based on the knowledge of all local verdicts and on the communication verification.

At step 1, we select a symbolic path issued from the symbolic execution of distributed systems. Concretely, either we can use the module TESTPURPOSE_SEL presented in Chapter 2 to verify the feasibility of a sequence of consecutive transitions, which is not immediate to obtain in the context of a distributed system, or we can use a module in Diversity called BEHAVIOR_SELECTION to cover an interesting behavior characterized by a sequence of non-consecutive transitions using the heuristics Hit-or-Jump (HoJ) [15]. From a methodological point of view, we will prioritize among all paths those that involve internal communications between subsystems to highlight causality relation and data identification constraints of internal communications. For this, we have implemented the Send Receive Pair Coverage criterion (SRPC) [68] as an exploration heuristics in Diversity which benefits from the module BEHAVIOR_SELECTION. Step 2 and 3 are applied to the global test purpose obtained from step 1.

At step 4, the overall testing process is initiated by a light coordination step to provide local test cases with the start moments of local test executions. Depending on initial actions of test case, a local test case knows when it has to emit or has to wait until an observation (reception on an internal channel or an emission), before possibly stimulating the LUT. This issue of controlling distant concurrent executions is known as the controllability problem in the context of distributed testing. A situation that alleviates this issue is the selection of a path that gives rise to a family of test purposes, defining only one test case in a situation to start with a stimulation.

At step 5, during test execution, a local test case can observe internal communications and provide inputs which are controllable to follow its local test purpose.

At step 6, we collect two local pieces of information: traces and verdicts. A local verdict results from the execution of a local test case on the localized subsystem. The local verdicts are likely to often be inconclusive because of numerous non-deterministic situations induced by the distributed context. A local trace of interactions between a subsystem with other subsystems (only internal communications) is also recorded. These pieces of information are then collected in order to produce a global test verdict, which consists of two stages:

- aggregation of the local verdicts
- communication correctness of the tuple of local traces with regard to a set of communication rules. The communication correctness is ensured whenever two following conditions hold:
  - There is enough emissions for receptions, i.e. the number of receptions of messages on a channel must be less than or equal to the number of emissions of messages on the same channel
  - The emission of an internal message must occur before its reception (i.e. causality relation $d(emission) \leq d(reception)$)

Given a subsystem $\mathbb{G}_i$ $(1 \leq i \leq n)$ in the distributed system $Sys = (\mathbb{G}_1, \ldots, \mathbb{G}_n)$, the global test verdict is determined based on the tuple of $n+2$ verdicts $(v_1, \ldots, v_n, v_{com\_int}, v_{com\_cau})$, where n first verdicts are local ones collected from local test cases whereas v_{com_int} and v_{com_cau} are respectively verdicts obtained from the verification of the first and second condition in the communication correctness. Concretely, the global test verdict is determined as follows:

- if all verdicts in the tuple are *Pass*, the global test verdict is *Pass*
- else if there exists at least one verdict in the tuple of $n+2$ verdicts which is determined as *Fail*, the global test verdict is *Fail*
- else, the global test verdict is *Inc*

For the global test purpose $p_{sys} = dsc_1 \dots dsc_m$ with $dsc_m = (dec_1, \dots, dec_n, dsc_{m-1})$, the tuple of sequences of transitions $(tr\text{-}seq(dec_1), \dots, tr\text{-}seq(dec_n))$ are qualified as *compatible* because they come from a common (satisfiable) symbolic path of the distributed system.

Each $tr\text{-}seq(dec_i)$ with $1 \leq i \leq n$ is a sequence of transitions characterizing a *local test purpose* tp_i. The subsystem LUT_i implementing $\mathbb{G}_i$ will be equipped with the associated test case LTC_i in charge of piloting LUT_i with the goal of following tp_i. The test case LTC_i pilots its localized subsystem LUT_i in the way that it computes stimulations based on previous observations in order to provide stimulations compatible with its local test purpose. This step is detailed by the construction rules presented in Section 3.3. Let us remark that for $tr\text{-}seq(dec_i) \in \{tr\text{-}seq(dec_1), \dots, tr\text{-}seq(dec_n)\}$, $tr\text{-}seq(dec_i)$ is a sequence of transitions of $\mathbb{G}_i$ that admits a symbolic counterpart that we note $SE(tr\text{-}seq(dec_i))$ in $Paths(\mathbb{G}_i)$. This means that $tr\text{-}seq(dec_i)$ admits at least one timed trace satisfying it. Therefore $tr\text{-}seq(dec_i)$ is feasible.

4.5.2 An ATM-Bank test purpose

In our approach, a global test purpose is characterized by a sequence of consecutive transitions that covers a particular scenario. In this illustration, we are interested in a blocking scenario caused by two consecutive withdrawals during some short period. This is a typical case of a stolen card and the thief wants to withdraw as much money as possible quickly. Concretely, the thief succeeds to withdraw quickly twice in a row (two executions of tr_5^a), the banker then assumes that it is abnormal. As a result, he decides to lock immediately the next transactions within some duration (transition tr_{10}^b). After this duration, the ATM receives the third request from the client which is locked due to the locking duration (transition tr_8^a). As we can see, to characterize this scenario, we easily know the transition of a withdrawal request and of a blocking request but it is not immediate to obtain the sequence of consecutive transitions in the distributed system Sys_{AB} since it involves many transitions in different subsystems and furthermore, we need to verify if the sequence is feasible.

To do this, we will take advantage of the module BEHAVIOR_SELECTION which allows to cover a coverage objective characterized by a sequence of non-consecutive transitions. The process of this module is presented in Figure 4.9. From a sequence of non-consecutive transitions in parameters $str'_1 = tr_3^a \cdot tr_2^b \cdot tr_5^a \cdot tr_3^a \cdot tr_2^b \cdot tr_5^a \cdot tr_{10}^b \cdot tr_3^a \cdot tr_2^b \cdot tr_8^a$ which contains 10 transitions, the module BEHAVIOR_SELECTION will compute a symbolic path in the symbolic execution tree of Sys_{AB} covering this sequence str'_1. We point out that the input sequence of transitions for BEHAVIOR_SELECTION is constructed based on the Send Receive Pair Coverage criterion (SRPC) [68] where a (sender, receiver) pair is a combination of two transitions labeled respectively by an internal emission and its corresponding internal reception. In case of str'_1, the (sender, receiver) pair (tr_3^a, tr_2^b) appears three times in the sequence to facilitate the coverage. For a more efficient coverage, during symbolic exploration, BEHAVIOR_SELECTION allows the reduction of an important number of symbolic paths. Indeed, with a special selection mechanism based on the heuristics Hit-or-Jump (HoJ) [15], BEHAVIOR_SELECTION performs the coverage by cutting off paths which are not related with our objective. At the end, the module BEHAVIOR_SELECTION produces the computed symbolic path (i.e. our global test purpose) which is characterized by a sequence of consecutive transitions.

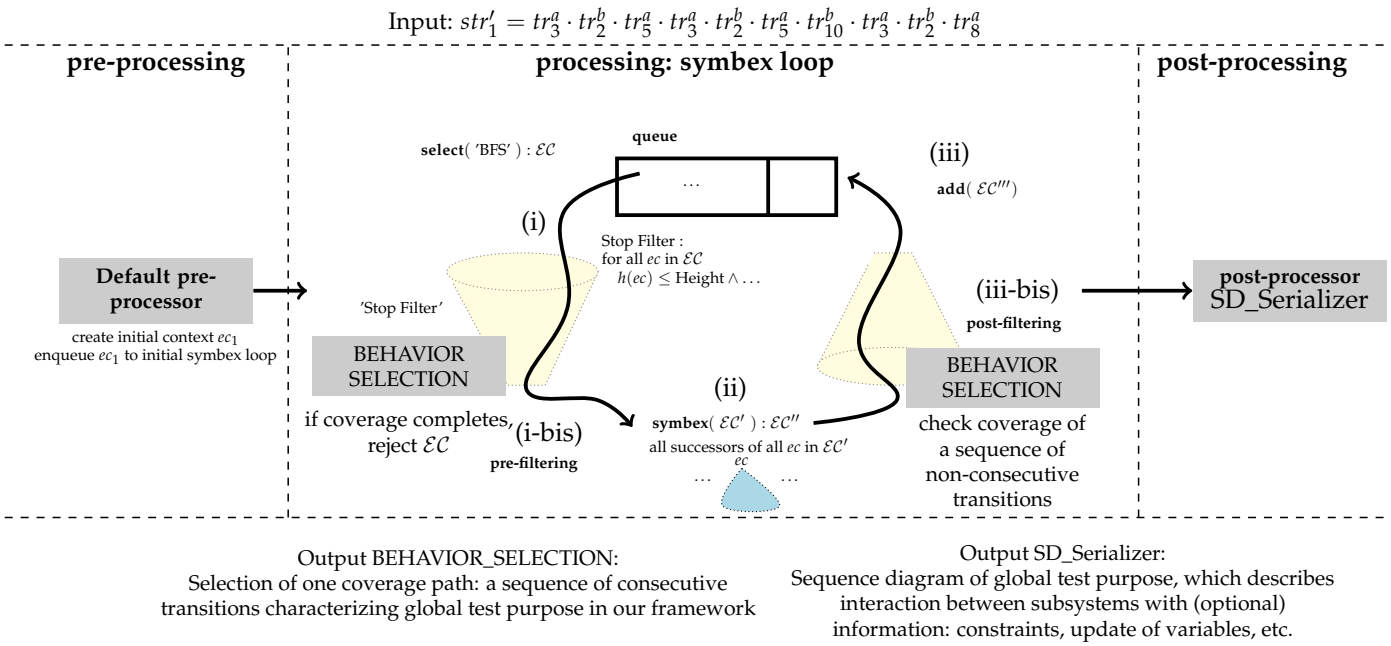


FIGURE 4.9: Module BEHAVIOR_SELECTION and module SD_Serializer

The sequence of consecutive transitions obtained from BEHAVIOR_SELECTION is presented in the following (23 transitions in total):

$$str_1 = tr^b_1 \cdot tr^a_1 \cdot tr^a_2 \cdot tr^a_3 \cdot tr^b_2 \cdot tr^b_3 \cdot tr^a_4 \cdot tr^a_5 \cdot tr^a_2 \cdot tr^a_3 \cdot tr^b_2 \cdot tr^b_3 \cdot tr^a_4 \cdot tr^a_5 \cdot tr^b_8 \cdot tr^b_9 \cdot tr^b_{10} \cdot tr^a_2 \cdot tr^a_3 \cdot tr^b_2 \cdot tr^b_4 \cdot tr^a_4 \cdot tr^a_8$$

The next step (i.e. step 2) consists in deriving canonically the tuple $(tr\text{-}seq(dec^m_a), tr\text{-}seq(dec^m_b))$ which represents respectively the sequence of consecutive transitions in ATM and Bank and they are determined as follows:

$$tr\text{-}seq(dec^m_a) = tr^a_1 \cdot tr^a_2 \cdot tr^a_3 \cdot tr^a_4 \cdot tr^a_5 \cdot tr^a_2 \cdot tr^a_3 \cdot tr^a_4 \cdot tr^a_5 \cdot tr^a_2 \cdot tr^a_3 \cdot tr^a_4 \cdot tr^a_8$$
$$tr\text{-}seq(dec^m_b) = tr^b_1 \cdot tr^b_2 \cdot tr^b_3 \cdot tr^b_2 \cdot tr^b_3 \cdot tr^b_8 \cdot tr^b_9 \cdot tr^b_{10} \cdot tr^b_2 \cdot tr^b_4$$

Until now, we presented step 1 and step 2 in our testing framework: the module BEHAVIOR_SELECTION has been applied to the sequence of non-consecutive transitions str'_1 in order to compute the sequence str_1 of consecutive transitions covering str'_1 and the tuple of sequences $(tr\text{-}seq(dec^m_a), tr\text{-}seq(dec^m_b))$ for subsystems. In the following, we present, in addition to TESTPURPOSE_SEL and TESTCASE_GEN, a new module called SD_Serializer [3] that we implemented as a serializer of symbolic tree and intervenes only in the step **post-processing** (Figure 4.9), allowing to visualize the global test purpose in the form of a sequence diagram. This visualization is very adequate to depict interactions between subsystems in distributed systems and facilitate the selection of the global test purpose. Therefore, it is very helpful for the first step in our testing framework.

SD_Serializer takes also in input the sequence of non-consecutive transitions str'_1. This module is a visitor of any symbolic tree and generates pieces of information in the textual entry language of PlantUML [4], which is an open-source tool allowing users to create visual diagrams from a plain text language. From the

[3]For more information about the implementation of SD_Serializer, refer to the page: https://projects.eclipse.org/proposals/eclipse-formal-modeling-project

[4]http://plantuml.com/

computed symbolic tree characterizing the global test purpose, this module retrieves traceability information such as external, internal messages, constraints, ... The generated sequence diagram is composed of many lifelines where each of them represents a subsystem in the distributed system. Moreover, we can annotate the lifelines with pieces of information such as time and data information, constraints related to execution of each subsystem, update of variables. All pieces of information are optional, which means Diversity's users are able to modify the information format or choose to display or not the information on the sequence diagram thank to dedicated configurable parameters.

Figure 4.10 shows the sequence diagram that is generated for the sequence str'_1. In the generated diagram, an output (or input) message is of the form $c!m@Z1$ (or $c?m@Z2$) where $Z1$ (or $Z2$) represents the emission (or reception) moment. In the path conditions for subsystems ATM and Bank which are detailed in the yellow boxes for each lifeline, we find again the causality relation and data identification constraints, such as: the causality `z_0 + z_1 <= z_0 + z_2` for the reception of the message on channel Init of ATM, the causality `z_0 + z_2 + z_3 + z_4 <= z_0 + z_1 + z_5` and data identification `(tid_1 = 1) && (x_1 = amt_1 + fee_0) && (aid_1 = ATM_ID)` for the reception of the message on channel Debit of Bank. We point out that for the causality relation $z_0 + z_1 \leq z_0 + z_2$, variable z_0 on the left (respectively on the right) represents the delay of Bank (respectively of ATM). And due to the mechanism of increase of index for creating new fresh variables by Diversity, the index of variables denoting duration $(z_0, z_1, z_2, \ldots)$ generated in path conditions are different with respect to the ones in subsection 4.4.3.

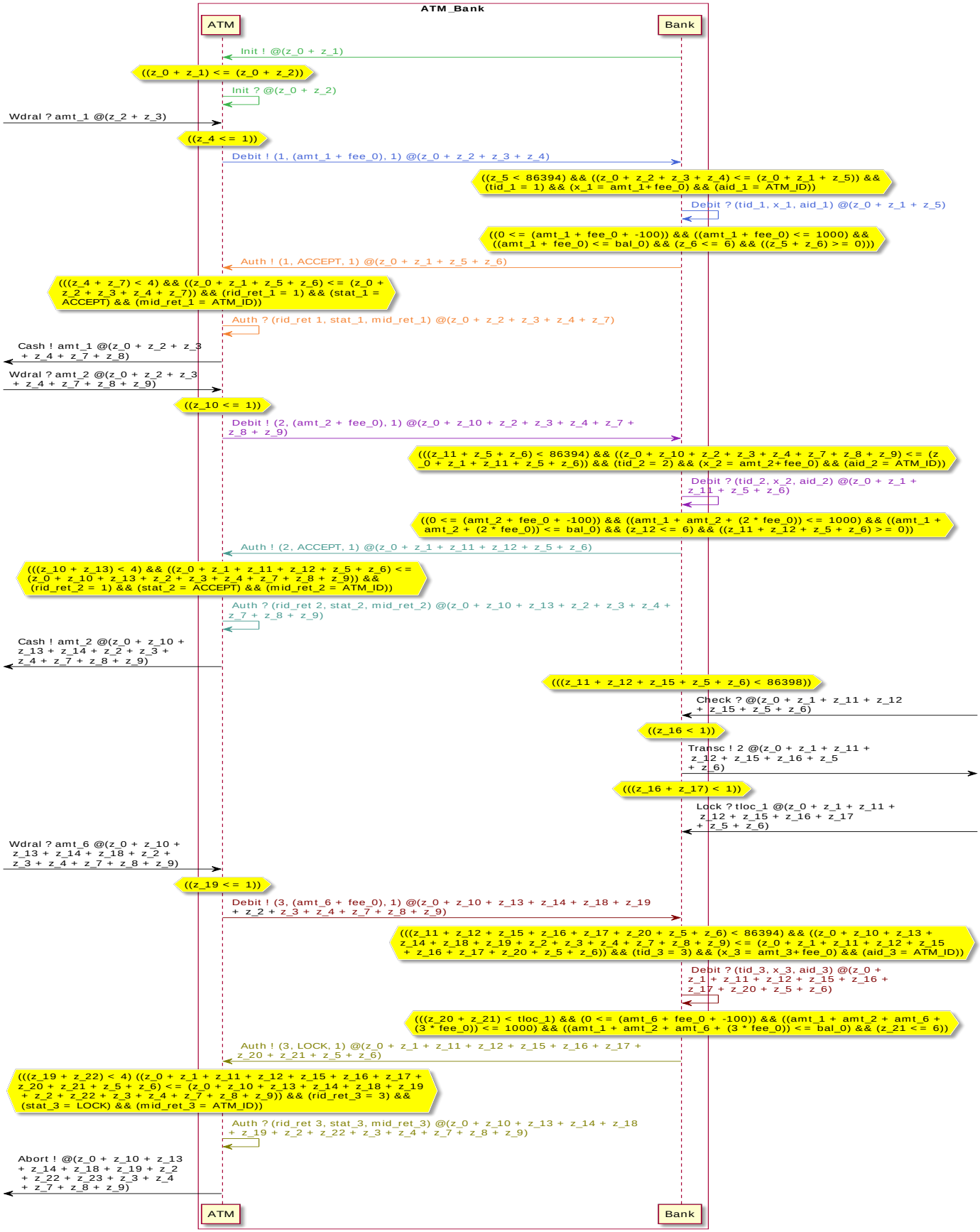


FIGURE 4.10: Scenario Locking request visualized by PlantUML resulted from coverage of global test purpose

4.6 Discussion of the approach

Our work is related to a family of model-based selection methods [12, 39, 68, 69, 13, 23, 14] which is dedicated to testing concurrent multi-process and/or distributed system and protocol models. These recommend to increase the coverage of synchronizations, pairwise communications and data exchange besides classical structural coverage criteria (e.g., all-states coverage, all-transitions coverage,...) which may fail short to select tests that exercise those. Our work is in the lineage of these works, in particular, we implement the *Send Receive Pair Coverage criterion (SRPC)* suggested by [68, 69] using symbolic execution techniques to compute the global test purpose, compositional models in which asynchronous communication is intertwined with complex time and data constraints, resorting to SMT solvers to check their satisfiability. Authors in [68, 69] use the reachability techniques offered by the tool UPPAAL [36] (models are networks of Timed Automata [66] under hand-shake synchronization) where time is symbolically handled as convex abstractions of clock values, known as zones [8] (pieces of data are rather enumerated). A significant benefit of our selection method is that it is fitted with an heuristic guided by an observer [2] allowing us to express reachability properties, in particular those which encode sequences of possibly non-consecutive pairwise emissions and receptions to cover. Our work is related in this aspect to [13, 23] which addresses the selection of scenarios using symbolic techniques and based on regular expressions describing sequences of (rather consecutive) synchronous operation calls in an untimed compositional model. When dealing with asynchrony, we believe that an heuristic approach can prove to be useful as reachability is undecidable in case of asynchronous message-passing, e.g., via unbounded fifo-queues.

4.7 Conclusion

This chapter presented a model-based distributed testing framework in reference models are n-tuple of TIOSTS that specify licit behaviors of the subsystems. The key problem addressed in this chapter is to automatically decompose a global test purpose defined as a scenario at the distributed system level, into a collection of n local test purposes, one for each subsystem. Those test purposes are meant to guide the local test case generation for the unitary subsystems that has been presented in the previous chapter. Therefore the local test cases are by construction coordinated in the sense that they will follow each a test purpose being part of a global scenario satisfying temporal and data correlation of internal communications. We have finally implemented a scenario generator in the Diversity tool which allows users to visualize the system scenarios in the user-friendly format of sequence diagrams. This visualization is more intuitive and suitable to inspect and analyze behaviors of distributed systems than the classical symbolic tree, which is more suitable for unitary systems.

Chapter 5

Models and scenario selection for information dissemination in Wireless Sensors Networks

Contents

In this chapter, we will illustrate our testing framework on a representative case study, even if it is still of reasonable size. In particular, it will be an opportunity to use our tools (design of timed models, asynchronous product of TIOSTS, symbolic execution techniques, projection of symbolic paths, dedicated coverage criteria, visualization of a distributed scenario considered as a global test purpose) in the context of the Diversity platform. A simplified version of the content of this chapter has been published in the following paper [61].

In this chapter, we develop behavioral models for a distributed IoT case study. The case study is a protocol for the dissemination of versioned fragments of firmware across a network of sensors (nodes) using the distributed Trickle algorithm [53]. Then, we experiment the scenario selection method and support tooling that we have presented in the previous chapter on the case study models. The chapter is structured as follows:

- Section 5.1 introduces briefly the context of the case study;
- In Section 5.2, we explain the principle of the distributed Trickle algorithm;
- Section 5.3 presents the Trickle-based *Multicast Protocol for Low-Power and Lossy Networks* (abbreviated by MPL);

- In Section 5.4, we develop behavioral models for the case study;
- In Section 5.5, we experiment the selection method on the developed models with respect to different network topologies. We have targeted the selection of scenarios achieving high-coverage of multicast communications reaching an up-to-date or outdated state(s) of sensor(s) in the network.
- Section 5.6 is the conclusion of the chapter.

5.1 Context

Internet of Things (IoT) designates the connection to the Internet of various devices (sensors, actuators, processors, etc.) evolving in a physical environment. Typical IoT applications allow the remote (via Internet) monitoring of household appliances, urban infrastructures, vehicle connectivity, agriculture production, and e-healthcare. Wireless Sensor Networks (WSN) are commonly part of such IoT applications. In fact, WSN are composed of a large number of sensors with limited resources (energy, memory, calculation, etc.) and called to operate independently during large periods (of the order of the year). To meet the new needs related to the evolution of their functionalities (new features, bug fix, security, etc.) and to remain compatible with other internet devices, it is necessary to propagate updates of binary firmwares [1] to be installed on the nodes (sensors) of the network. These sensors networks have variable size and topology (hundreds or thousands of sensors, which can be added or removed, and possibly mobile). In addition, the dissemination of information across the network is through sensor-to-sensor short-range communications since such small devices can be equipped only with small radio antennas. Such communications may be asymmetric as well: a node A may send messages to a node B without the opposite being possible. Therefore, efficient update protocols for all sensors firmware in a network have to meet several objectives:

- reduce the number and the size of packets (messages) exchanged between sensors to save the sensor batteries. In particular, when the sensors all have the same version of the firmware (consistent network), then the frequency and size of the messages exchanged must be minimal: message exchanges should be used to ensure that all sensors share same firmware version. Due to the changing topology of the network, firmware update is a continuous activity, constantly iterated: nodes must periodically communicate to find out if there is a new version of the firmware;
- quickly propagate new firmware versions to minimize periods when sensors do not share the same versions. When sensors discover that neighboring nodes need updates, the latest version of the firmware must be released quickly.

Trickle [53, 54, 51] is the state-of-the-art distributed algorithm for the dissemination and update of information across a WSN. This algorithm is provided as a standard library in TinyOS [52] and Contiki [27], two of the best known firmware Operating Systems (OS) for WSN. Trickle is also used in recently standardized WSN protocols namely the Multicast Protocol for Low Power and Lossy Networks (MPL) [38] and the Routing Protocol for Low Power and Lossy Networks (RPL) [86].

We choose the MPL protocol as a case study in particular when used to disseminate (firmware) upgrades in WSN. On the difference of RPL multicast (which is not dissemination oriented), MPL is well adapted to disseminate fragmented firmware (as a set of smaller-size packets) which is most efficient in such energy-constrained WSN (e.g. as discussed in [47]). We develop for this case behavioral models based on the reference specification of the protocol [38]. An MPL implementation is already available in Contiki OS [27].

[1] Firmwares are specific user software applications communicating directly with the device hardware and installed into the device non-volatile flash memory which makes them more adaptable.

5.2 Trickle algorithm

As glimpsed in previous section, a WSN network can be seen as a directed graph connecting nodes to their neighbors which can be reached by their transmissions, i.e., broadcasts of messages, without the need of routing via other intermediate nodes. Usually the dissemination of an information by a given sender node to all nodes in the network cannot be granted by a single message transmission. Trickle is a fully distributed algorithm where each node applies a set of rules according to its state, i.e., the information it holds, with the objective of converging towards a global stable state of the network where all nodes have the same information. The Trickle algorithm can be described as follows [53, 54, 51, 58]:

- each node maintains a current interval τ, a counter c and a broadcasting time t in current interval τ,
- global parameters to all nodes (same values) are k the redundancy constant, τ_l the lowest value for τ and τ_h the highest value for τ,
- each node applies the following rules:

  1. at the start of a new interval a node resets its timer and counter c and sets at random t to a value in $[\tau/2, \tau[$,
  2. if the node receives a message consistent with the information it holds, it increments c,
  3. when its timer reaches t, the node broadcasts the message carrying the information it holds if $c < k$,
  4. when its timer expires at τ, it increases its interval length by setting τ to $\min(2 \cdot \tau, \tau_h)$ and starts a new interval,
  5. when a node receives a message that is inconsistent with its own information, then if $\tau > \tau_l$ it sets τ to τ_l and starts a new interval, otherwise it does nothing.

Each time an inconsistency is detected, the value τ is set to τ_l, then τ is doubled up to τ_h. This makes known the "cost" per inconsistency in terms of transmissions number $\approx log(\tau_h/\tau_l)$: at most one transmission per τ-interval, occurring exactly at t. We note that the node transmits only if its neighbors are unlikely to be up-to-date, when $c < k$ given c counts receptions of consistent messages in the interval (k is fixed based on number of neighbors). Moreover, since small intervals are considered immediately after the inconsistency, the frequency of transmissions is greater at the beginning (and decreases when approaching τ_h), which allows nodes to quickly share the same information. Now nodes are not necessarily synchronized, yet Trickle suggests choosing a random t (in $[\tau/2, \tau[$) together with imposing a listen-only period (first half of τ) in order to enhance the distribution of the transmission load between nodes in the interval (and hence energy costs).

5.3 Trickle algorithm in MPL protocol

The *Multicast Protocol for Low-Power and Lossy Networks* (MPL) [38] uses Trickle to disseminate two kinds of messages:

- *data message* containing functional data disseminated by nodes in the network
- *control message* used to notify all neighbors of the most recently received data message

The protocol associates each data message with a sequence number and a unique identifier of the original sender node which generates the message (seed in MPL parlance). These pieces of information are not modified by all intermediate nodes transmitting the message. During the operation, nodes maintain a lower-bound

sequence number that represents the sequence number of the oldest data message the nodes are willing to receive or transmit. In such setting, control messages are used to inform neighbors about recently received data messages. MPL specifies two dissemination strategies:

- *proactive forwarding* which uses Trickle to schedule transmissions of data messages a limited number of times without knowing if the neighbors have the message or not;
- *reactive forwarding* which uses Trickle to handle the forwarding of control messages (a limited number of times as well), these messages allows data messages to be scheduled only if neighbors are not up-to-date.

The case study considers reactive forwarding which is specific to the protocol MPL.

5.4 Behavioral models for MPL protocol

In this section, we present the full model of the case study. Some modeling facilities that are naturally encoded in the input language of Diversity have been allowed and discussed. This is to present a readable form of TIOSTS forming the MPL processes of the case study system, and especially keep the intuition on the Trickle algorithm being implemented by each of those TIOSTS.

5.4.1 Processes of sensor nodes architecture

Figure 5.1 depicts the architecture of the MPL processing units per node in case of reactive forwarding: each node is defined by $n+1$ processes, managed each by a Trickle algorithm. Process cp (resp. dp_i, $1 \leq i \leq n$) handles the dissemination of all control messages (resp. the dissemination of exactly the i^{th} data message identified by its sequence number). Each process is a TIOSTS. As suggested in the figure we use a *template* to instantiate many times the same TIOSTS, those on their turn can be grouped to form the enclosing template of a node, this hierarchical template mechanism is supported by the Diversity tool. For example, the TIOSTS ControlMsgProcess and DataMsgProcess which are respectively depicted in Figure 5.2 and Figure 5.3 are generic templates which are instantiated to specify an enclosing node template, namely Node. The latter is instantiated on its turn to obtain nodes $n_1, n_2, \ldots$ (see Figure 5.1). Communication between nodes is specified by asynchronous message-passing between their respective processes (over unbounded fifo) where sending a message is not blocking for the sender process and messages are received in the same order that they are issued, the kind of communications defined in Chapter 4.

We distinguish a particular node n_1 in the architecture which holds the data messages to be disseminated (a.k.a data message generator). As we can see, n_1 can send data to n_2 but the opposite is not possible. We suppose that in general, each node contains the following sub-processes: one ControlMsgProcess and many DataMsgProcess. Moreover, the ControlMsgProcess of n_2 receives directly control messages disseminated by n_1 on channel ControlMsg, whereas each process of data messages of n_1 can disseminate data to all processes of data messages of n_2 through channel DataMsg, i.e. the dissemination of data messages from dp_1 of n_1 is received by all sub-processes $dp_1, \ldots, dp_n$ of n_2.

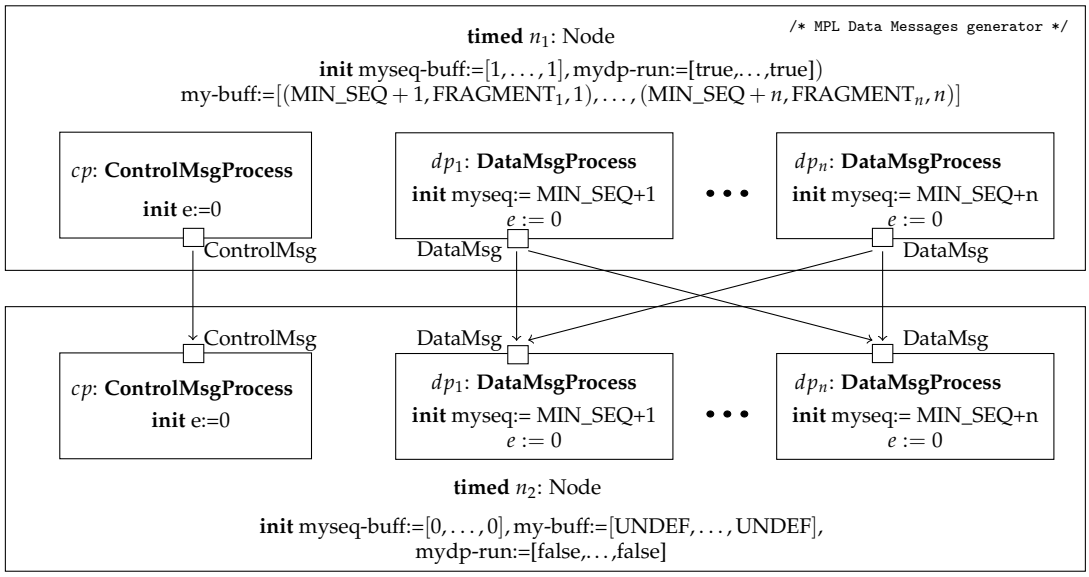


FIGURE 5.1: Nodes sub-processes instantiation.

5.4.2 Compact automata for process modeling

The (template) TIOSTS that we discuss for MPL modeling are of two kinds: one for handling control messages (Figure 5.2), the other for handling data messages (Figure 5.3). Yet both have similar structure as each implements the Trickle algorithm (presented informally in Section 5.2). In order to compact and simplify the structure of those TIOSTS, we use transitions with sequential statements which are naturally encoded by the Diversity tool's entry language xLIA. Statements include communication actions act which are either of the form $c?x$ that denotes the reception, on channel c, of a value which is stored in variable x, or of the form $c!t$ that denotes the emission, on channel c, of the value represented by term t; assignments $x := t$ which assigns the variable x with a value denoted by t; **newfresh**(x) randomly assigns x with a new fresh variable; guards on variables $[\phi]$ are used to denote firing conditions of transition. More generally, statements (stm) are also built considering the null statement (**skip**) and the following control primitives: sequence (;), condition (**if** ... **then** ... **else** ...) or counted-repetition (**for**$(nat\ i = N : M)$ ... with N, M, i natural numbers and $N \leq M$).

A transition will be of the form $q \xrightarrow{stm,\mathbb{K}} q'$ where q is a source state, q' is a target state, stm is a statement, and $\mathbb{K}$ is a subset of clocks to reset. Whereas, transitions in theory (see Section 2.2 of Chapter 2) have the form $(q \xrightarrow{\mathbb{K},\phi,act,\rho} q')$ where $\mathbb{K}$ is as previous a subset of clocks to reset, ϕ is a firing guard, act is a communication action (_ denotes the absence of such communication action), ρ is a substitution of variables (other than clocks) which represents their updates with new values.

Therefore, with notation of Chapter 2, two consecutive transitions for example are respectively of the form:

$$cp_2 \xrightarrow{\emptyset\ ,\ cl{=}\tau\ ,\ \_\ ,\ [\tau{:=}min(2{\cdot}\tau,\tau_k),\ \mathbf{newfresh}(t),\ c{:=}0,\ e{+}{+}]} cp_2^i$$

$$cp_2^i \xrightarrow{\{cl\}\ ,\ cl{=}\tau\wedge\tau/2{\leq}t{<}\tau\ ,\ \_\ ,\ [\ ]} cp_1$$

N.B., here we consider the case when timer τ expires, then τ is doubled and a new t is chosen within the new half of the τ-interval (see Trickle rules in Section 5.2).

Those subsequent transitions are equivalent to following transition using sequential statements:
$cp_2 \xrightarrow{cl=\tau;\ \tau:=min(2\cdot\tau,\tau_k);\ \textbf{newfresh}(t);\ \tau/2\leq t<\tau;\ c:=0;\ e{+}{+}],\ \{cl\}} cp_1$ (see Figure 5.2).
Condition and repetition statements allows us as well to have this compact form of transitions, rather than making explicit two (or more) transitions for a decision and so on.

5.4.3 Control message process

Figure 5.2 depicts the ControlMsgProcess which handles control messages using the Trickle algorithm.

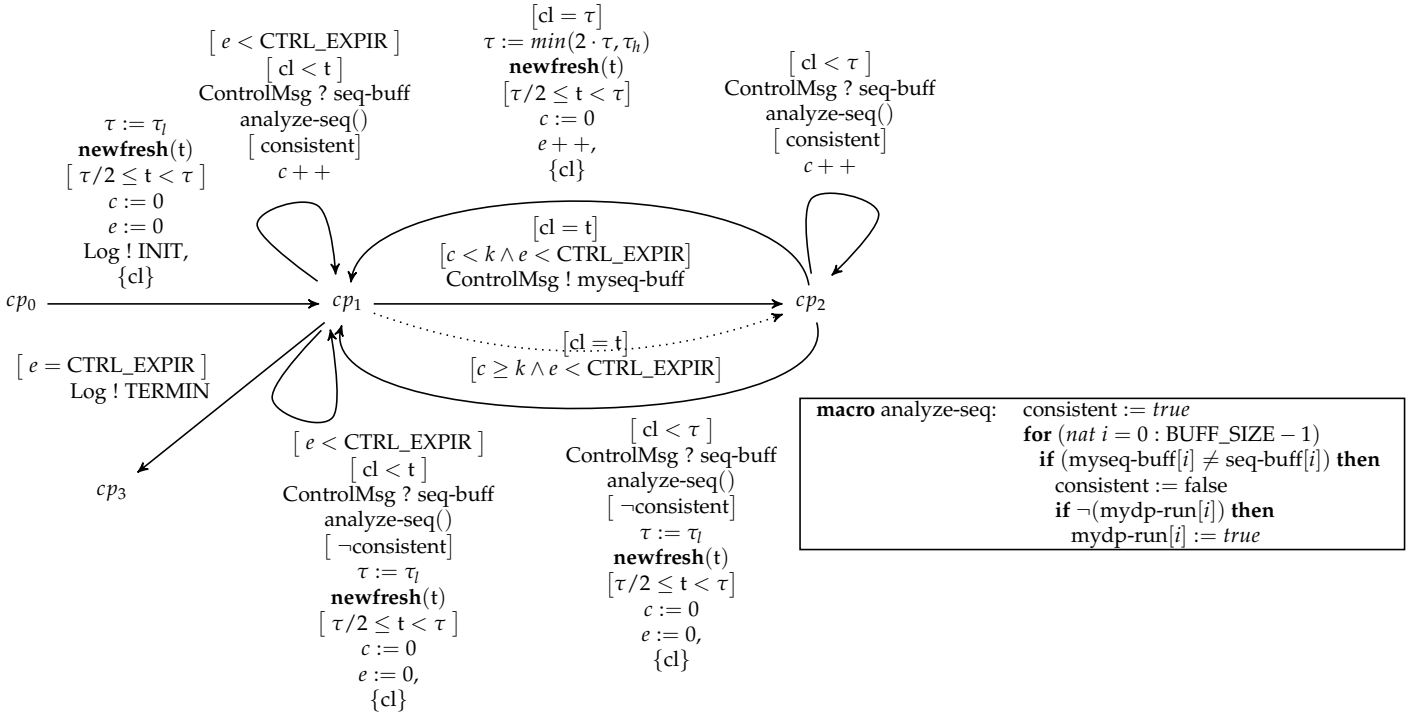


FIGURE 5.2: Control messages processing.

From the initial state cp_0, the control messages process assigns t with a new fresh value using the action newfresh(t), this new value for t is constrained by the guard $\tau/2 \leq t < \tau$. The clock cl is initially reset, as glimpsed before it is used to activate the different Trickle events, that is when reaching t and starting new τ-intervals.

When the process reaches the state cp_1, it can receive a control message from a neighboring node before reaching t (loop-transitions on cp_1). Such message carries a sequence of 0 and 1. The value 1 (resp. the value 0) at place $i \geq 0$ of the sequence indicates that the neighbor holds (resp. does not hold) the data message which is identified by sequence number MIN_SEQ $+ i$. The message is processed as follows (see macro analyse-seq): for each value of the sequence on which the node and its neighbor does not agree, the corresponding data process of the node (the instance of DataMsgProcess which initialized with $myseq :=$ MIN_SEQ $+ i$, see Figure 5.1) is started by assigning mydp-run$[i]$ with *true*. This allows us a) in case of a not up-to-date neighbor, to schedule Trickle transmissions; and b) in case of the node itself not being up-to-date, to wait for upcoming data messages, yet without starting transmissions. Trickle rule related to (in)consistency is then applied: if the neighbor is up-to-date, i.e., the node and its neighbor agree on all values of the sequence,

then the counter c of process is incremented (upper loop-transitions on cp_1), otherwise a new τ-interval is considered for the process where τ and c are reset, since additional little-spaced control messages are needed for convergence (lower loop-transitions on cp_1). When cl = t, the transmission is scheduled only if c < k in order to inform neighbors of data messages the nodes possess (upper transition $cp_1 \rightarrow cp_2$), otherwise the process does nothing (lower transition $cp_1 \rightarrow cp_2$).

At state cp_2, while $cl < \tau$ and received control messages are consistent, the counter c is incremented (loop-transition on cp_2), upon the reception of inconsistent control message the process starts a new τ-interval by resetting both cl and c (lower transition $cp_2 \rightarrow cp_1$). When $cl = \tau$, the process increases the listening interval by doubling τ (up to τ_h), and that is when the variable e is incremented (upper transition $cp_2 \rightarrow cp_1$).

The variable e disables the overall Trickle behavior when it reaches CTRL_EXPIR, and terminates the process (transition $cp_1 \rightarrow cp_3$).

5.4.4 Data message process

The DataMsgProcess is given in Figure 5.3. The process implements the Trickle algorithm and therefore is structured in a similar manner as the control messages process. We recall that it is instantiated as many times as the number of the data messages to be disseminated, the variable myseq of the process identifies its associated data message.

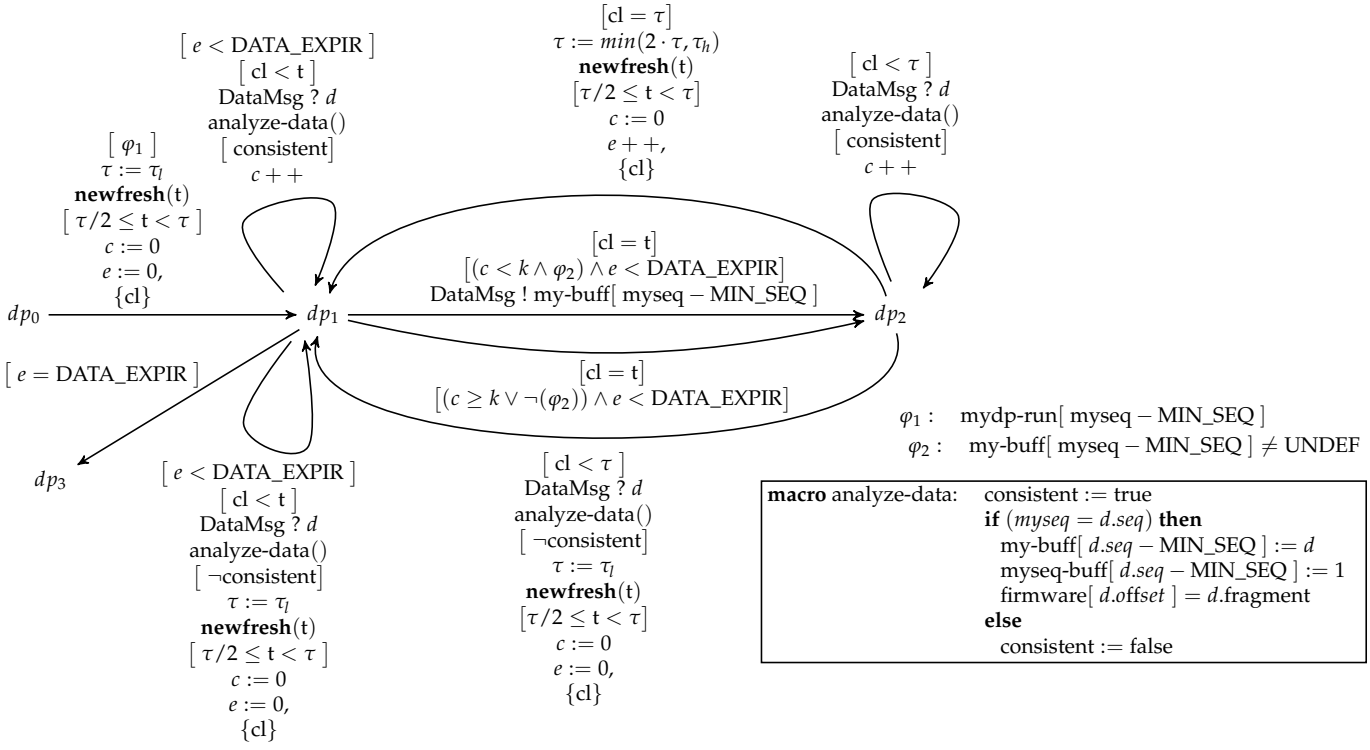


FIGURE 5.3: Data messages processing.

When φ_1 : mydp-run[myseq − MIN_SEQ] is evaluated to *true*, the process is activated (transition $dp_0 \rightarrow dp_1$). The process can transmit the data message if it exists in the node buffer namely my-buff, i.e., φ_2 : my-buff[myseq − MIN_SEQ] ≠ UNDEF is *true* (upper transition on $dp_1 \rightarrow dp_2$). Actually, there are two possible ways to hold such a message: a) either at instantiation, this is the case of the instance $n_1 : Node$ of

Figure 5.1 (my-buff[myseq − MIN_SEQ] is properly initialized) as n_1 represents the data messages generator in that setting; b) or upon its reception from a neighboring node (upper loop-transitions respectively on dp_1 and dp_2). In the latter case, the process stores the message in its buffer, it also write the carried firmware fragment at the appropriate location of the flash memory (see macro analyse-data). Firmware reconstitution is not part of the MPL protocol, naturally it is just a well-known application of this multicast protocol usage in WSN, which constitutes our case study.
As for the control messages process, the process is associated with a Trickle timer expirations limit DATA_EXPIR monitored by its variable e which is incremented at each τ-interval expiration (upper transition $dp_2 \rightarrow dp_1$) and is reset when τ is assigned with τ_1. The process terminates when e reaches DATA_EXPIR (transition $dp_1 \rightarrow dp_3$).

5.5 Experimenting with scenario selection

In this section, we propose firstly a selection method in Diversity to compute feasible symbolic paths which cover a sequence of elements (formula, action). Then, we present several WSN network topologies and two coverage objectives in a WSN network to cover by using proposed selection method. The last part is dedicated to the analyses of scenarios which are generated from the selection method for two coverage objectives.

5.5.1 An extended selection method with observers

We recall that the idea is to guide the symbolic exploration with the objective to compute feasible system paths covering sequences of (sender, receiver) pairs implementing the *Send Receive Pair Coverage criterion (SRPC)* [68, 69]. Those are combination of two transitions labeled respectively by an internal emission and its corresponding internal reception where an emission ($c!m$) of a message m by some subsystem is followed by the corresponding reception ($c?m$) of m by another subsystem (see Section 4.5.2).

The system is asynchronous, meaning that emissions and receptions are likely to be separated with potentially many other intertwining actions which are difficult to guess beforehand, that's why we propose to use an heuristic combined with observers:

- A parametric heuristic Hit-or-Jump [15] (implemented in Diversity by the module BEHAVIOR_SELECTION, see Section 4.5.2) is used to cope with the potential combinatorial explosion due to asynchrony. It runs a regular breadth-first exploration of the symbolic tree up to a certain height (H: height of exploration), and after that, if a path satisfies the coverage the exploration stops, otherwise a number of paths (HC: hit count) which maximize the coverage are chosen at random to re-iterate with another breadth-first (remaining paths are disregarded). The number of carried breadth first is bounded as well (JTL: jump trials limit), when all paths in a given breadth-first do not cover anything, a number of paths can be chosen at random to continue the exploration (JC: jump count);
- A synchronous observer [2] to express reachability of states (targets of some feasible system paths) after some sequences of non-consecutive pairwise emissions/receptions. It is able to read I/O actions and to progress as the actions of the sequence are covered, unmatched in-between actions are ignored up to a parametric limit (if the limit is reached the path is disregarded). Moreover, the observer can monitor formulas on process's clocks and data variables which need to be satisfied by some encountered states in the exploration.

This selection is integrated with a scenario generator from the selected system paths (see module SD_Serializer, Section 4.5.2). Those are produced in the form of sequence diagram annotated with time and data information and can be visualized by the PlantUML tool (`http://plantuml.com`).

We present next the WSN topologies of nodes for the case study. For those, we will present experimental results on scenario selection using the tooling support discussed above.

5.5.2 Some sensor nodes topologies

For each topology, we illustrate the connections between two different nodes by arrows. A connection from the node n_i to the node n_j in the network is drawn as in Figure 5.4. Whenever n_j is reached by n_i, we call that n_j is a neighbor of n_i and it can receive messages which are emitted by n_i since it belongs to the wireless range of n_i. However, n_i cannot receive anything from n_j since it does not belong to the wireless range of n_j. We recall that the connection between n_i and n_j is not bidirectional.

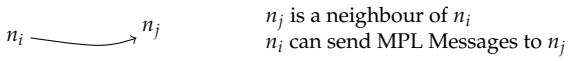


FIGURE 5.4: A connection in the network

Let us introduce the first topology T1 which is composed of three nodes n_1, n_2 and n_3 where n_1 is defined as the MPL Data Message generator which receives all fragments of the firmware from an external source and emits them to all nodes in the network. In the sequel, without loss of generality, we consider always that n_1 is a data message generator for each topology.

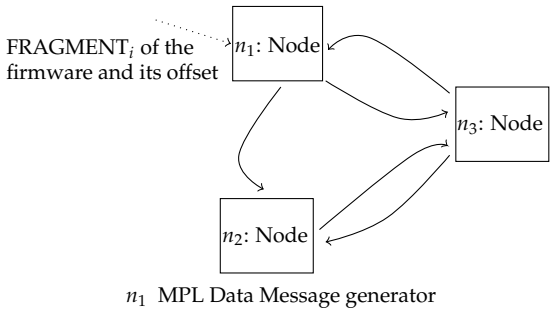


FIGURE 5.5: Topology T1

In our experimentations, the number of nodes in the topologies are progressively increased. The second and the third topology (denoted respectively by T2 and T3) are depicted in Figure 5.6. It consists of 4 nodes namely n_1, n_2, n_3 and n_4. The only difference between T2 and T3 is that in T2, the node n_3 can send its data to the node n_2 whereas in T3, this connection is not possible.

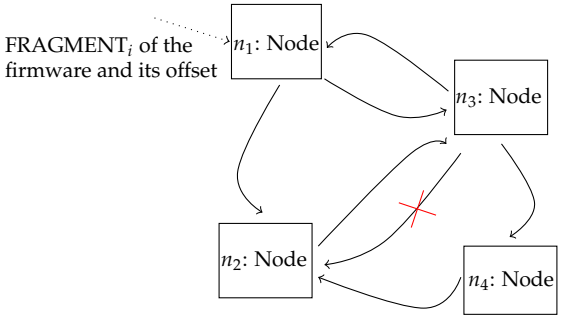


FIGURE 5.6: Topology T2 and T3 (removing the connection from n_3 to n_2)

The next topology that we consider in our experimentations is T4. The latter has one more node than the two previous topologies. This node (i.e. n_5) is reached by the connection from node n_3 to n_5.

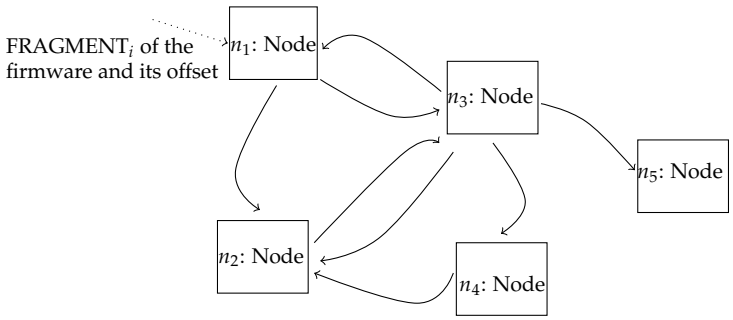


FIGURE 5.7: Topology T4

5.5.3 Coverage objective 1 - All nodes are updated

We consider next the selection of system paths (or scenarios) which represent situations of all nodes being up-to-date, in other words they all hold the newest data messages in their buffer at a certain point of the exploration. The experiments are conducted by using firstly **an observer** encoding a sequence of formulas on state variables to be satisfied by the selected path in the order of their occurrence in the sequence, then sequences mixing actions and formulas. The latter being more expressive allows in general faster selection when combined with the heuristic as we will see next.

5.5.3.1 Selection using sequences of ordered formulas

The sequence SEQ1. For the first experiments, we use a sequence composed of formulas in order to express our coverage objective *all nodes are up-to-date*:

SEQ1: sequence of ordered formulas f_i with $1 \leq i \leq 3$ and f_i is defined as n_i.myseq-buff $= [1, 1]$

which requires three nodes in the topology to be updated. Indeed, when a node receives all fragments which are necessary for the firmware update, its buffer myseq-buff is expected to be assigned with $[1, 1]$. For the topology T1 which is composed of 3 nodes, the sequence SEQ1 is concretely the following:

$f_1 : n_1$.myseq-buff $= [1, 1]$
$f_2 : n_2$.myseq-buff $= [1, 1]$
$f_3 : n_3$.myseq-buff $= [1, 1]$

The sequence SEQ1 is constructed based on the topology: as we know, n_1 is considered as the MPL Data Message generator, this node is always updated first of all, lately n_2 and n_3 are updated by receiving data from n_1.

Table 5.1 shows results for the experiment conducted on topology T1. Each correspond to an heuristic exploration that has been executed with 20 trials, and the metrics are such that number of evaluations, time and rate, are computed on average for the 20 trials.

The coverage result for SEQ1 is shown in Table 5.1. It is clear that the coverage ratio has grown (from 60% to 80% and 90%) with the increase of the local height (respectively from 2 to 3 and 4). However, the algorithm takes a longer coverage duration, from 19 seconds to more than 2 minutes and especially, more than 12 minutes in case of the local height of 4. A more important number of evaluations is also observed (respectively from 274 to 1903 and 10287).

This increase is explained as follows: with a greater local height, the heuristic explores locally all possible sequences of a bigger height, which takes obviously more time than the execution of a smaller one. Furthermore, we obtain a better coverage ratio since a greater local height explores longer paths than a smaller local height, which increases the chances to cover the considered elements in the sequence.

Property	Hit-or-Jump parameters				#EV	Time	Rate
	#H	#JTL	#HC	#JC			
All updated	2	40	2	2	274	19s120ms	60%
All updated	3	30	2	2	1903	2m25s	80%
All updated	4	30	2	2	10287	12m9s800ms	90%

#H: height of exploration triangle, #JTL: jump trials limit,
#HC: hit count, #JC: jump count
#EV: count of successful evaluations of exercised transitions
successful ratio for 20 trials of the selection heuristics
Experimentations on an ASUS computer, memory 8GB, processor Intel Core i7 2.00GHz $\times$ 4, Ubuntu operating system.

TABLE 5.1: Experimentations with SEQ1 on topology T1

The sequence SEQ2. The sequence SEQ1 is slightly modified in order to obtain the sequence SEQ2 as follows:

$f_{1,0} : n_1.\text{myseq-buff}[0] = 1$ $\quad$ $f_{1,1} : n_1.\text{myseq-buff}[1] = 1$
$f_{2,0} : n_2.\text{myseq-buff}[0] = 1$ $\quad$ $f_{2,1} : n_2.\text{myseq-buff}[1] = 1$
$f_{3,0} : n_3.\text{myseq-buff}[0] = 1$ $\quad$ $f_{3,1} : n_3.\text{myseq-buff}[1] = 1$

The formula $f_{i,j}$ means that the node n_i has its array myseq-buff whose value at the position j is 1, i.e. the node n_i has already received the data message of firmware at the position j.
This new sequence allows the heuristic to improve the coverage ratio compared with the SEQ1, the exploration checks the formula myseq-buff $= [1, 1]$ for each node, i.e. the node must receive two data messages of firmware to satisfy the given formula. Whereas in SEQ2, this formula is rewritten by two separate formulas myseq-buff$[0] = 1$ and myseq-buff$[1] = 1$, giving a sequence of 6 formulas for 3 nodes in total. Obviously, it is easier to cover sequentially two separate formulas than their conjunction.

According to the experiments, it takes more than 18 seconds to update the whole topology on average (line 1 in Table 5.2). With a greater local height (i.e. 3), it takes only 1m35s to generate the scenario, which is much faster than the experimentation with the same configuration for SEQ1. Similarly, for the height of 4, the sequence SEQ2 takes only 7 minutes to cover the scenario compared to 12 minutes of the sequence SEQ1.

Once again, we find that the strategy using the sequence SEQ2 takes more time to generate scenarios with a greater local height. This result is similar to the experiments for SEQ1 for the same reasons.

Moreover, for the last experimentation where the local height of the breadth-first trials is 4, we have an absolute coverage ratio (20 successes over 20 times). We realize that by increasing the local height, we can increase the chance to cover the scenario, but the coverage duration can also grow very quickly.

Selection objective	Hit-or-Jump parameters				#EV	Time	Rate
	#H	#JTL	#HC	#JC			
All updated	2	40	2	2	234	18s110ms	90%
All updated	3	30	2	2	1142	1m35s	90%
All updated	4	30	2	2	5727	7m41s790ms	100%

TABLE 5.2: Experimentations with SEQ2 on topology T1

Sometimes, even though the exploration is executed with an important local height, we cannot generate the scenario. Indeed, since the symbolic execution chooses to explore randomly a number of symbolic paths, it may not choose the right paths to continue the process and does not succeed in covering the sequence.
Comment. From the experiments with SEQ1 and SEQ2, we can draw the following remark: the sequence SEQ2 performs a better coverage result than SEQ1. In the next experiments, our sequences will be constructed in a similar way as SEQ2.

The sequence SEQ3. Now, let us consider the coverage of the objective *all nodes are up-to-date* on the topology T2 and T3 by using the sequence of formulas SEQ3 defined in the same way as SEQ2.

SEQ3: the sequence of ordered formulas $f_{i,j}$ with $1 \leq i \leq 4, 0 \leq j \leq 1$ and $f_{i,j}$ is defined as n_i.myseq-buff$[j] = 1$

Table 5.3 summarizes the experimental results. For a topology composed of 4 nodes (T2 and T3), we realize that the exploration needs more time to cover the sequence SEQ3 in all cases: for example, when the local height is 2, the execution needs on average 18s110ms to cover the objective in the topology T1, whereas it needs more than four times to cover the objective in T2 and T3 (respectively 1m20s and 1m18s). Or in case where the local height is 4 (line 3 and line 6), the exploration takes almost a third of an hour to update completely the four nodes. This can be due to the size of topology of T2 and T3, as they have one more node than T1, more exchanges to be updated. Once again, we find that the coverage ratio for the two topologies is increased with the growth of the local height as in the first topology T1.

Selection objective	Hit-or-Jump parameters				#EV	Time	Rate
	#H	#JTL	#HC	#JC			
All updated (T2)	2	40	2	2	689	1m20s135ms	20%
All updated (T2)	3	30	2	2	4279	4m28s400ms	70%
All updated (T2)	4	30	2	2	22290	19m49s60ms	80%
All updated (T3)	2	40	2	2	796	1m18s250ms	20%
All updated (T3)	3	30	2	2	4144	3m42s	90%
All updated (T3)	4	30	2	2	19572	17m25s120ms	80%

TABLE 5.3: Experimentations with SEQ3 on topology T2 and T3

Comment. Until now, our selection heuristic shows good execution time: certainly we do not conduct exhaustive exploration but we have succeeded in achieving the coverage, within a reasonable time if we take as reference running model-checking techniques on similar number of parallel Trickle processes, see works [26, 87, 88], objective *all nodes are up-to-date*. We remind the reader that a node in the MPL protocol is defined by 3 Trickle processes, which gives in total 12 processes for a topology of 4 nodes, the heuristic spent slightly more than 17 minutes to achieve the coverage with a very high coverage ratio (80%).

Selection objective	Hit-or-Jump parameters				#EV	Time	Rate
	#H	#JTL	#HC	#JC			
All updated	2	40	2	2	1009	1m46s906ms	10%
All updated	3	30	2	2	8807	12m12s160ms	60%
All updated	4	30	2	2	45120	37m15s60ms	60%

TABLE 5.4: Experimentations with SEQ4 on topology T4

The sequence SEQ4. Let us consider now the topology T4 which consists of 5 nodes (see Figure 5.7) with the sequence of formulas SEQ4.

SEQ4: the sequence of ordered formulas $f_{i,j}$ with $1 \leq i \leq 5, 0 \leq j \leq 1$ and $f_{i,j}$ is defined as n_i.myseq-buff$[j] = 1$

With a local height of 2, the exploration reaches only 2 successful coverage over 20 trials (Table 5.4). The performance is worse than the topologies T1, T2 and T3, due to the increased size of topology T4. When increasing the local height of the breadth-first we obtain more interesting results in terms of coverage ratio, however it takes more than 12 minutes to reach the coverage for a local height of 3, and more than half an hour (37 minutes) for a local height of 4.

5.5.3.2 Selection using sequences mixing actions and formulas

Previously, we have used selection guided by sequences of ordered formulas. In the following, we propose to guide the selection of sequences mixing formulas and actions. We target as before the coverage where *all nodes are up-to-date*. We compare also the experiment results obtained by this selection method with the ones using the sequences of formulas only (from previous subsection).

The sequence SEQ5. The following sequence is defined for the topology T3:

SEQ5: $(n_1, \text{ControlMsg}![1,1])$.
$(n_2, \text{ControlMsg}?[1,1]).(n_3, \text{ControlMsg}?[1,1])$
$(n_1, \text{DataMsg}!*).(n_2, \text{DataMsg}?*).(n_3, \text{DataMsg}?*)$.
$(n_3, \text{DataMsg}!*).(n_4, \text{DataMsg}?*)$.
$(n_3, \text{DataMsg}!*).(n_4, \text{DataMsg}?*)$.
$\bigwedge_{i \leq 4} n_i$.myseq-buff $= [1,1]$

The sequence SEQ5 denotes a natural idea to target the update of the most distant node (n_4) from the generator (n_1) in T3. In order to quickly start the dissemination, the sequence suggests that n_1 sends a control message ControlMsg![1, 1] to n_2 and n_3 to start their data messages processes: myseq-buff of n_2 (resp. of n_3) is expected to be assigned with $[0, 0]$ (if nothing happens in between). We note that n_1 has already started its data messages process due to initialization of mydp-run := $[\text{true}, \ldots, \text{true}]$ (see Figure 5.1). SEQ5 suggests using few other data messages exchanges to reach the coverage objective more rapidly. On the other hand, the formula expresses that all nodes are up-to-date which represents the main goal, i.e., when it is satisfied the heuristics asserts.

Selection objective	Hit-or-Jump parameters				#EV	Time	Rate
	#H	#JTL	#HC	#JC			
All updated	2	40	2	2	167	16s200ms	90%
All updated	3	30	2	2	870	38s250ms	100%
All updated	4	30	2	2	1826	1m12s210ms	100%

TABLE 5.5: Experimentations with SEQ5 on topology T3

The coverage results are depicted in Table 5.5. We also show the graphical depiction of a scenario for this topology in Figure 5.8, which has been generated by our module SD_Serializer from a system path satisfying the coverage objective. By comparing with the results obtained in Table 5.3, we can see that shorter time is needed to obtain the coverage for all considered local heights. Moreover, the coverage ratio is much better by using this selection.

Comment. By applying this kind of sequence to the topology T3, we obtain a better result than using the sequences composed of formulas only, both in term of coverage ratio and duration. However, this selection requires that the user understands the topology in order to provide a compatible sequence. For example, in topology T3, a sequence that proposes the update of n_4 before n_2 and n_3 is impossible, since n_4 is the most distant node from the generator.

5.5.4 Coverage objective 2 - At least one node is outdated

Let us highlight now an atypical, less studied, scenario of MPL protocol (Figure 5.9) which can give insights on the unfairness of the diffusion algorithm under some type of topology. The scenario has been computed for topology T3 (Figure 5.6) by using the following sequence which aims to reach a situation in which the most distant node n_4 from the generator n_1 cannot be updated.

SEQ6 : $[0](n_1, \text{Log!INIT}); [0](n_2, \text{Log!INIT}); [0](n_3, \text{Log!INIT}); [0](n_4, \text{Log!INIT});$
$[0](n_1, \text{ControlMsg!}[1,1]); [0](n_2, \text{ControlMsg?}[1,1]); [0](n_3, \text{ControlMsg?}[1,1]);$
$[2](n_1, \text{DataMsg!}(1, \text{FRAGMENT}_0, 0));$
$[2](n_2, \text{DataMsg?}(1, \text{FRAGMENT}_0, 0)); [2](n_3, \text{DataMsg?}(1, \text{FRAGMENT}_0, 0));$
$[4](n_4, \text{ControlMsg!}[0,0]); [4](n_2, \text{ControlMsg?}[0,0]]);$
$[8](n_3.dp_1, \text{Log!TERMIN}); [4](n_4.dp_1, \text{Log!TERMIN})$
$[n_4.\text{my-buff} = [\text{UNDEF}, \text{UNDEF}]];$
MPL parameters: $k=2$, $\tau_l=4$, $\tau_h=8$, Control_EXPIR$=2$, DATA_EXPIR$=2$.

Limits on possible in-between actions ($L=0,2,4,8,\ldots$) are the result of an iterative process starting with small values with the objective to achieve the targeted coverage as soon as possible. This limit is determined as the value inside the brackets between two any communication actions. The sequence SEQ6 imposes a strict sequencing of initialization messages of control processes for all nodes to quickly start the dissemination ($L=0$). Then, it expresses that generator n_1 (initially possesses all the fragments of the firmware) broadcasts its control message to its neighbors n_2 and n_3. Few more actions are then indicated to better guide the heuristic.

A computed scenario is depicted in Figure 5.9, the heuristics has explored more behaviors than in the first scenario (of Figure 5.8). This is due to the non-trivial Trickle situation of the node n_4 in topology: n_4 can only receive messages from n_3, which can be in a situation where it cannot send data messages anymore to its neighbors (including n_4) after receiving "enough" consistent messages (redundancy constant k has been reached according to Trickle algorithm). So the heuristics has generated significantly more exchanged messages compared to the first scenario and of course, more transitions being evaluated (see the evaluation number in the table).

Selection objective	Hit-or-Jump parameters				#EV	Time	Rate
	#H	#JTL	#HC	#JC			
At least one node outdated	2	40	2	2	167	40s160ms	90%
At least one node outdated	3	30	2	2	870	2m15s	100%

TABLE 5.6: Experimentations with SEQ6 on topology T3

5.5.5 Analyses of generated scenarios

In this section, we discuss the two scenarios that have been previously referred to, and analyze them with respect to the MPL/Trickle required behavior.

5.5.5.1 Scenario for the coverage objective *all node are updated*

The scenario of Figure 5.8 is depicted with the following graphical conventions: each emission of a given message and its corresponding reception are colored with the same unique color in the diagram. Besides, each message emission (or reception) by a node has a label of the form $c!m@Z$ (or $c?m@Z$) where Z is a local timestamp (to the node) as a sum of fresh variables among z_0, z_1, z_2, ... Fresh variables occurring in a local timestamp definition represent durations of past executions of the node transitions (including the emission

or reception transition being considered), as glimpsed in Section 5.4. In the scenario, initial durations from the beginning for n_1, n_2, n_3, n_4 are respectively denoted by z_0, z_1, z_2, z_3.

In the scenarios generation, we have used Diversity to check the feasibility of all causal communications of the form $(n_i, c!m@Z_1)$, $(n_j, c?m@Z_2)$, $(n_j, d!m@Z_3)$ and $(n_i, d?m@Z_4)$ using those timestamps: we check that the formula $Z_1 \leq Z_2 \wedge Z_3 \leq Z_4$ (by construction we have $Z_1 < Z_4$ and $Z_2 < Z_3$) in conjunction with local PC of the nodes are satisfiable[2]. Actually, the PC are generated from the successive symbolic evaluation of guards on clocks and other owned variables occurring in a given path. For illustration, we give next the PC generated by the heuristics for the node n_3.

Conditions on clocks of n_3:

$$\text{PC11} \begin{cases} z_{186} + z_{198} + z_{302} + z_{310} + z_{338} = t_{54} \\ z_{198} + z_{302} + z_{310} + z_{338} + z_{378} + z_{530} = t_{171} \\ z_{310} + z_{338} + z_{378} = t_{279} \\ z_{198} < t_{171} \\ z_{310} < t_{279} \\ z_{82} < t_{15} \end{cases}$$

Conditions on other variables of n_3:

$$\text{PC12} \begin{cases} t_{15} < 4 \\ t_{54} < 4 \\ t_{171} < 4 \\ t_{279} < 4 \\ 0 \leq (t_{15} + -2) \\ 0 \leq (t_{54} + -2) \\ 0 \leq (t_{171} + -2) \\ 0 \leq (t_{279} + -2) \end{cases}$$

with $k = 2$, $\tau_\text{l} = 4$, $\tau_\text{h} = 8$, DATA_EXPIR $= 2$ and CTRL_EXPIR $= 2$.

PC11 and PC12 reflects the intended transmissions and receptions dates of both control and data messages during the successive τ-intervals for n_3 (with $\tau_\text{l} = 4$). For example, the transmission of the control message ControlMsg!$[1,1]@z_{186} + \ldots + z_{82}$ occurs exactly at $z_{186} + \ldots + z_{82}$ with $t_{54} = z_{186} + z_{198} + z_{302} + z_{310} + z_{338} \wedge 2 \leq t_{54} < 4$ where t_{54} is a fresh variable which denotes the random value chosen for t (constrained by $\tau/2 \leq t_{54} < \tau$ in the system). The same analysis applies on the last transmission timestamp of the data message DataMsg!$(1, \text{FRAGMENT}_0, 0)@z_{186} + \ldots + z_{82}$ with $t_{171} = z_{198} + z_{302} + z_{310} + z_{338} + z_{378} + z_{530}$. The latter achieves the update of the node n_4, see message DataMsg?$(1, \text{FRAGMENT}_0, 0)@z_3 + \ldots z_{675}$ (being out-of-reach of the node n_1, the data messages generator n_1).

Table 5.7 gives the coverage achieved by this first scenario with respect to the Send Receive Pair Coverage criterion (SRPC) [68] "In order to achieve full sr-pairs coverage every sr-pair must be executed at least once in testing" where an sr-pair is a combination of send and receive statements for a message m in different machines, which are respectively of the form $(n_i, c!m)$ and $(n_j, c?m)$ in our setting. Most of the missing sr-pairs have been separately achieved by producing trivial sequences for the heuristics: regarding the model, we will not have all the coverage since n_1 can only send the control message ControlMsg!m_1 with $m_1 = [1,1]$, other values for m_1 are not activated in this model. This is an intrinsic limitation to this coverage criterion.

[2]In order to check satisfiability of formulas, Diversity uses off-the-shelf SMT solvers such that Z3, CVC4, etc.

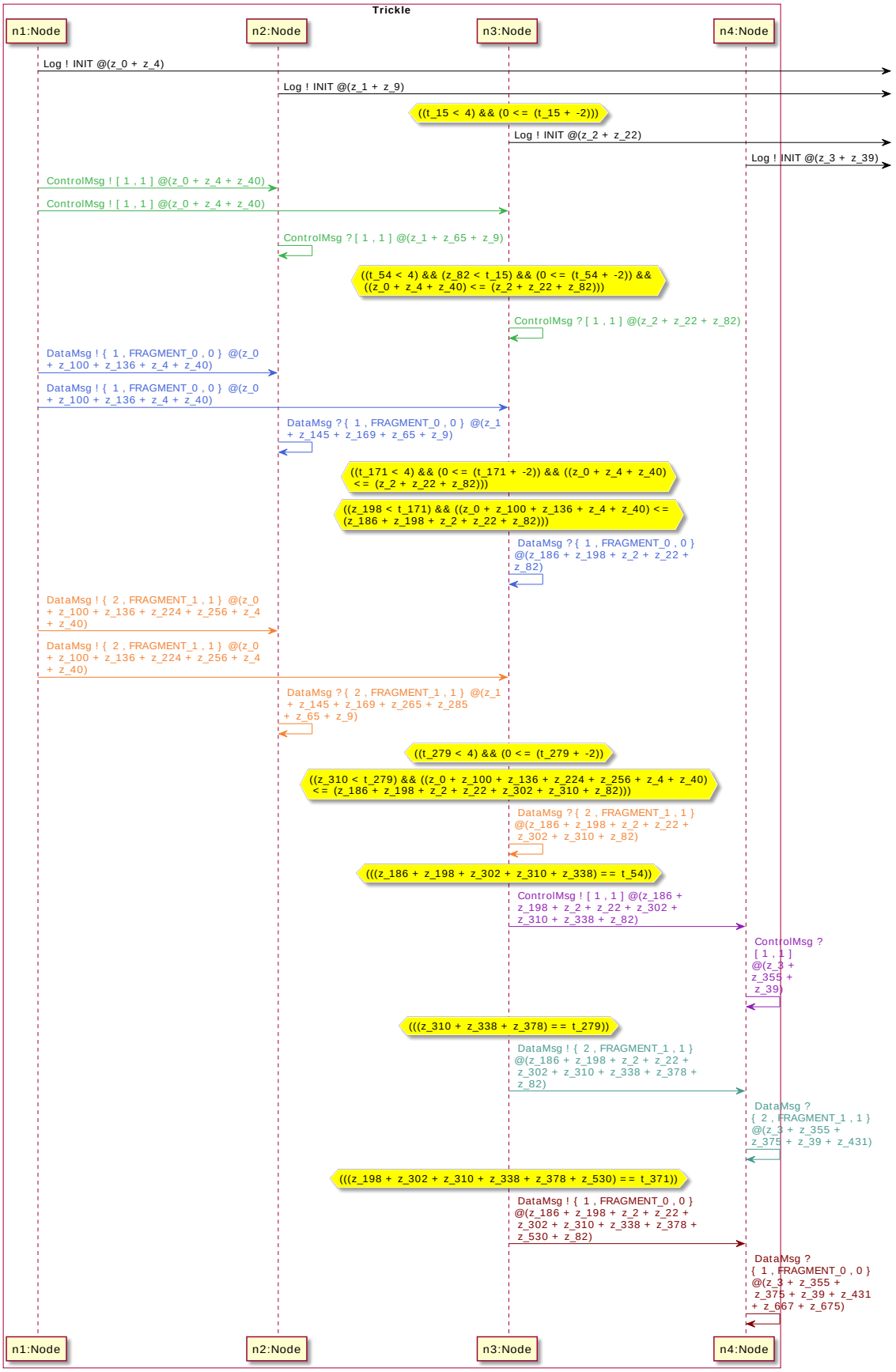


FIGURE 5.8: Scenario for the first coverage objective - All nodes are updated

sr-pair	send, receive statements	Coverage
sr(1,2)(m_1)	(n_1, ControlMsg!m_1), (n_2, ControlMsg?m_1)	1/4
sr(1,2)(m_2)	(n_1, DataMsg!m_2), (n_2, DataMsg?m_2)	2/2
sr(1,3)(m_1)	(n_1, ControlMsg!m_1), (n_3, ControlMsg?m_1)	1/4
sr(1,3)(m_2)	(n_1, DataMsg!m_2), (n_3, DataMsg?m_2)	2/2
sr(2,3)(m_1)	(n_2, ControlMsg!m_1), (n_3, ControlMsg?m_1)	none
sr(2,3)(m_2)	(n_2, DataMsg!m_2), (n_3, DataMsg?m_2)	none
sr(3,1)(m_1)	(n_3, ControlMsg!m_1), (n_1, ControlMsg?m_1)	1/4
sr(3,1)(m_2)	(n_3, DataMsg!m_2), (n_1, DataMsg?m_2)	none
sr(3,2)(m_1)	(n_3, ControlMsg!m_1), (n_2, ControlMsg?m_1)	1/4
sr(3,2)(m_2)	(n_3, DataMsg!m_2), (n_2, DataMsg?m_2)	none
sr(3,4)(m_1)	(n_3, ControlMsg!m_1), ($n4$, ControlMsg?m_1)	2/4
sr(3,4)(m_2)	(n_3, DataMsg!m_2), ($n4$, DataMsg?m_2)	2/2
sr(4,2)(m_1)	(n_4, ControlMsg!m_1)(n_2, ControlMsg!m_1)	none
sr(4,2)(m_2)	(n_4, DataMsg!m_2)(n_2, DataMsg!m_2)	none

$m_1 \in \{[0,0],[1,0],[0,1],[1,1]\}$
$m_2 \in \{(1, \text{FRAGMENT}_0, 0), (2, \text{FRAGMENT}_1, 1)\}$

TABLE 5.7: Achieved coverage by the (1st) scenario of Figure 5.8 with respect to Send Receive Pair Coverage criterion (SRPC) [68].

5.5.5.2 Scenario for the coverage objective *at least one node is outdated*

In the scenario of Figure 5.9, initial durations from the beginning for n_1, n_2, n_3, n_4 are respectively denoted by z_0, z_1, z_2, z_3. PC of processes of n_3 (an updated node) are reported for illustration. Those reflect the moment of transmissions and receptions of both control messages and data messages within chosen τ-intervals (with $\tau_1 = 4$): e.g., the reception of the first control message ControlMsg?$[1,1]$@$(z_2 + z_{22} + z_{86})$ occurs exactly at $z_2 + z_{22} + z_{86}$ with $z_{86} < t_{15}$ where t_{15} denotes the random value chosen for t (constrained by $\tau/2 \leq t_{15} < \tau$), meaning that the reception of this message satisfies the constraint $\text{cl} < t$. Before this reception, the clock cl of the control messages process (of n_3) has been reset after initialization (Log!INIT@$(z_2 + z_{22})$). Subsequently, at the instance t_{58}, n_3 disseminates its control message that has just been updated under the condition $z_{186} + z_{198} + z_{302} + z_{318} + z_{338} = t_{58}$. Now the node n_3, despite of being already updated, cannot send data messages to its neighbors, including n_4, since the counter c of its first data process has reached the redundancy constant k (equals to 2) after receiving two times the data message respectively from n_1: DataMsg?$(1, \text{FRAGMENT}_0, 0)$@$(z_2 + \ldots + z_{198})$ and n_2: DataMsg?$(1, \text{FRAGMENT}_0, 0)$@$(z_2 + \ldots + z_{470})$. For this data message, n_4 terminates the corresponding handling process (Log?TERMIN@$(z_3 + \ldots + z_{1063})$) at the end of the scenario after two consecutive τ-intervals without being updated by its neighbor n_3. Such unfair situations arise from the value of redundancy constant k being common to all nodes and hence should be set carefully according to the nodes topology.

For more information about MPL models and experimentations, please visit this GitHub repository: https://github.com/ngo-minh-thang-nguyen/MPL_Trickle

5.6 Conclusion

In this chapter, we have provided behavioral models for the Trickle-based MPL protocol specifying licit timed multicast communications which can be used in Model-based Testing activities. Then, in order to automate the selection of relevant scenarios to be tested from such distributed models with high combinatorial of communications (due to asynchrony), we have extended the exploration heuristic by observers. Those encode coverage sequences of non-consecutive pairwise emissions and receptions, including predicate formulas on the content of messages. Our selection method and support tooling allowed us to generate in reasonable time relevant scenarios for the Trickle-based MPL protocol displaying an up-to-date or outdated states of network.

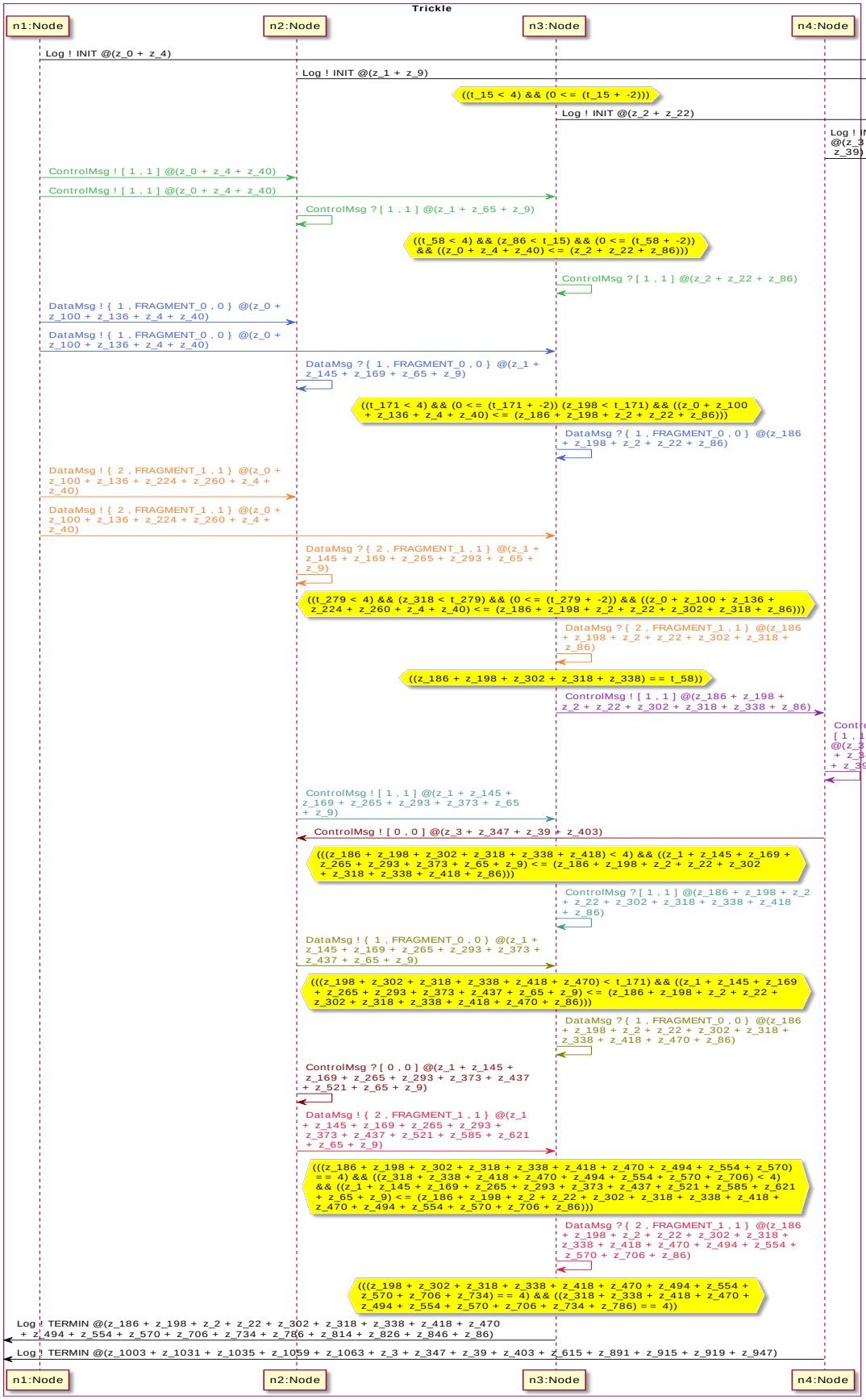


FIGURE 5.9: Scenario for the second coverage objective - At least one node is outdated

Chapter 6

Conclusion

Contents

6.1 Summary

Distributed systems (DS) consist of a number of independent subsystems, executing concurrently on different machines and interacting, by message passing, through a communication network so to complete a common objective. Such kind of systems are subjected to errors of different nature including errors related to excessive delays in replying to a received message or errors due to misplaced/erroneous data content for a received message. In order to capture, hence prevent the malicious effect of such errors, testing of DS becomes a vital pre-deployement task. Because of the peculiarities of DS (distributed nature, the lack of a global clock, non-deterministic behaviors), testing of DS turns out to be non trivial.

In the context of testing distributed systems, this dissertation develops new insights by proposing a novel testing framework with regard to the issue of test data generation. The presented approach belongs to the family of Model-Based Testing approaches, i.e. those approaches where the specification of the System Under Test (SUT) and of what should be considered a correct implementation is formally described. We stress that our approach is not concerned with so-called *oracle problem* which consists of checking the traces to detect non-conformance between the SUT and the model using a mathematical conformance relation (for us tioco). Indeed, we have deliberately based our contribution in the context of the framework of distributed system modeling and testing described in [32, 10], which is very focused on the issue of conformance and its verification: in a few words, given a set of traces under the form of sequences of actions and durations, one per remote subsystem, the approach described in [32],[10] recommends analysing each of the traces in an off-line mode up to the tioco conformance relation (using for example [5]) and the tuple of traces with regard to some communication rules (using constraint solving techniques). Since the approach of [32], [10] has the advantage of being compositional and facilitating the resolution of the oracle problem, our modeling assumptions for distributed systems will be exactly the same. However, [32], [10] left aside the question of generating test data, whether in terms of managing test data or defining test selection criteria appropriate for DS. We therefore sought to complete the work of [10] with test generation capabilities, while keeping exactly the same test architecture, i.e. one test case per remote subsystem, to benefit from the same definitions and properties relating to conformance.

In this context, in Chapter 2, we introduce the TIOSTS formalism that we have adopted for the purpose of modeling timed reactive systems. TIOSTS formalism makes it relatively easy to specify properties about data and time. Moreover the formalism has the key advantage of being equipped with symbolic execution

mechanisms. The main interest of this chapter is to set the main definitions and notations that will be used throughout the manuscript, to introduce small illustrative examples, to present the customizable Diversity platform allowing users to define their own functionalities about models and exploration strategies of their symbolic execution trees. In connection with the following chapters, the notion of deterministic models is precisely expressed in relation to properties of symbolic execution trees: this will be useful because it is expected that TIOSTS defining test cases are deterministic. For the same reasons, the facilities of the Diversity platform to target user-defined objectives are illustrated through a simple example, that of following a sequence of transitions from the initial model.

In Chapter 3, we detail the construction of an online unitary test case, i.e. a test case with the ability to compute the next test data taking into account the previous reactions of the SUT. The main difficulty is to combine constraints on data and time, the possible occurrence of message arrivals on internal channels as well as the need to follow a test purpose defined as the form of a finite path of the symbolic execution tree. Let us recall here that the particularity of our approach for managing unitary test case design is to integrate that local systems SUT are tested in the context of a distributed system, i.e., in the context where third parties interact with the SUT through internal channels, without control on the part of the tester. The construction of the test case as a TIOSTS is defined using rules covering the different cases likely to occur depending on the progress in the test purpose, the reception on the internal channels, ... Rules are in the form of a TIOSTS transition so that, by taking the test purpose as a guiding skeleton, it is possible to represent the test case as a rather particular TIOSTS (in the form of a tree with verdicts on the leaves). The implementation in Diversity is done in two steps, the first to build a symbolic execution tree of the reference model including the considered test objective, and the second to build transitions whose source nodes are those of the test purpose, by applying the rules.

In Chapter 4, a distributed system is specified as a collection of unitary subsystems given as TIOSTS. System behaviors are captured by the symbolic execution of the whole system, defined by intertwining of symbolic executions of subsystems and modeling of asynchronous internal communications using waiting queues. Each symbolic path of the whole system defines by projection a family of local paths that can serve as local test purposes. The causality relation and the data identification are taken into account during the projection mechanism in order to provide local testers with these constraints that need to be satisfied by subsystems. As a part of our contribution, to provide the user with an alternative tool for the visualization of a global test purpose (besides the symbolic tree), we implemented in the Diversity tool a textual generator allowing for visualizing the global test purpose in the form of a sequence diagram annotated with temporal and data constraints reflecting the feasibility of the test purpose.

In Chapter 5, we illustrate our testing method with a case study which is a Wireless Sensor Network based on the communication protocol MPL. The objective of the protocol is to update network nodes (i.e. devices) by means of the Trickle algorithm built for the MPL protocol. From the specification of the protocol, we constructed timed symbolic models. As part of our contribution, we implemented selection criteria in the Diversity tool that allow the generation of global test purposes which are relevant for testing from models. The interesting properties of the network that are expressed in a high level language, such as, e.g. *all devices are updated or at least one device is outdated*, can be considered as a global test purpose. With the aim of experimenting the selection criteria, their practical usage has been then highlighted by using the Diversity tool for generating different scenarios displaying up-to-date or outdated state of devices for many network topologies.

6.2 Future research directions

A natural continuation of our work on unitary testing is to transform the generated test cases into an executable format (e.g., as TTCN [85] scripts) and to define an adequate test instrumentation (e.g., using the test execution tool Titan [75] for TTCN) which allows: i) handling the asynchronous arrival of internal messages and systematically associate them with timestamps, and ii) integrating solvers that will be called to calculate the next stimulations and analyze the conformance of the observations. We point out that the speed of the instrumented test case including the solvers calls and the speed of the SUT should be carefully studied in order to ensure reliable testing.

We have carefully designed local test cases with the challenge to follow as far as possible the test purpose, in the possible presence of internal messages that may not be observed as expected, and thus lead to dedicated inconclusive verdicts. We take benefit of the compositional result of [32] ensuring that the verification that subsystems communicate properly, together with our online test cases for subsystems according to tioco are sufficient to assess the conformance of the global distributed system to its model. To limit inconclusive situations (mainly due to intrinsic controllability difficulties of distributed testing), our framework can be extended firstly by identifying new criteria on tuples of local test purposes while ensuring a better controllability, and secondly by considering coordination messages between remote test cases.

Acronyms

BFS **B**readth-**F**irst **S**earch

DFS **D**epth-**F**irst **S**earch

DS **D**istributed **S**ystem

DSC **D**istributed **S**ystem **C**ontext

DUT **D**istributed System **U**nder **T**est

EC **E**xecution **C**ontext

ENV **E**nvironment

FSM **F**inite **S**tate **M**achine

ioco **i**nput **o**utput **co**nformance relation

IOLTS **I**nput **O**utput **L**abelled **T**ransitions **S**ystems

IOSTS **I**nput **O**utput **S**ymbolic **T**ransition **S**ystems

IoT **I**nternet **o**f **T**hings

LAN **L**ocal **A**rea **N**etworks

LTC **L**ocal **T**est **C**ase

LTS **L**abelled **T**ransitions **S**ystems

LUT **L**ocalized System **U**nder **T**est

MBT **M**odel-**B**ased **T**esting

MPL **M**ulti-cast **P**rotocol for **L**ow-Power and **L**ossy Networks

PC **P**ath **C**ondition

pioco **P**robabilistic **i**nput **o**utput **co**nformance relation

pIOTS **P**robabilistic **I**nput **O**utput **T**ransition **S**ystem

RFS **R**andom-**F**irst **S**earch

SE **S**ymbolic **E**xecution

SRPC **S**end **R**eceive **P**air **C**overage

STS **S**ymbolic **T**ransition **S**ystems

SUT **S**ystem **U**nder **T**est

TA **T**imed **A**utomata

tioco **t**imed **i**nput **o**utput **co**nformance relation

TIOSTS **T**imed **I**nput **O**utput **S**ymbolic **T**ransition **S**ystems

TSTS **T**imed **S**ymbolic **T**ransition **S**ystems

WSN **W**ireless **S**ensor **N**etworks

xLIA **x**ecutable **L**anguage for **I**nteraction and **A**ssemblage

Bibliography

[1] W. L. Andrade et al. "Abstracting Time and Data for Conformance Testing of Real-Time Systems". In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. Mar. 2011, pp. 9–17. DOI: 10.1109/ICSTW.2011.82.

[2] Étienne André. "Observer Patterns for Real-Time Systems". In: *2013 18th International Conference on Engineering of Complex Computer Systems*. July 2013, pp. 125–134. DOI: 10.1109/ICECCS.2013.26.

[3] Mathilde Arnaud, Boutheina Bannour, and Arnault Lapitre. "An Illustrative Use Case of the DIVERSITY Platform based on UML Interaction Scenarios". In: *Electr. Notes Theor. Comput. Sci.* 320 (2016), pp. 21–34. DOI: 10.1016/j.entcs.2016.01.003. URL: https://doi.org/10.1016/j.entcs.2016.01.003.

[4] Boutheina Bannour, Christophe Gaston, and David Servat. "Eliciting Unitary Constraints from Timed Sequence Diagram with Symbolic Techniques: Application to Testing". In: *18th Asia Pacific Software Engineering Conference, APSEC 2011, Ho Chi Minh, Vietnam, December 5-8, 2011*. 2011, pp. 219–226. DOI: 10.1109/APSEC.2011.40. URL: https://doi.org/10.1109/APSEC.2011.40.

[5] Boutheina Bannour et al. "Off-Line Test Case Generation for Timed Symbolic Model-Based Conformance Testing". In: *Testing Software and Systems - 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings*. 2012, pp. 119–135. DOI: 10.1007/978-3-642-34691-0_10. URL: https://doi.org/10.1007/978-3-642-34691-0_10.

[6] Clark Barrett et al. "CVC4". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV'11. Springer-Verlag, 2011, pp. 171–177. ISBN: 978-3-642-22109-5. URL: http://dl.acm.org/citation.cfm?id=2032305.2032319.

[7] Roman Barták, Miguel A. Salido, and Francesca Rossi. "Constraint satisfaction techniques in planning and scheduling". In: *Journal of Intelligent Manufacturing* 21.1 (2010), pp. 5–15. ISSN: 1572-8145. DOI: 10.1007/s10845-008-0203-4. URL: https://doi.org/10.1007/s10845-008-0203-4.

[8] Gerd Behrmann et al. "Lower and upper bounds in zone-based abstractions of timed automata". In: *International Journal on Software Tools for Technology Transfer* 8.3 (2006), pp. 204–215. ISSN: 1433-2787. DOI: 10.1007/s10009-005-0190-0. URL: https://doi.org/10.1007/s10009-005-0190-0.

[9] Axel Belinfante. "JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Javier Esparza and Rupak Majumdar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 266–270. ISBN: 978-3-642-12002-2.

[10] Nassim Benharrat et al. "Constraint-Based Oracles for Timed Distributed Systems". In: *Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9-11, 2017, Proceedings*. 2017, pp. 276–292. DOI: 10.1007/978-3-319-67549-7_17. URL: https://doi.org/10.1007/978-3-319-67549-7_17.

[11] Henrik Bohnenkamp and Axel Belinfante. "Timed Testing with TorX". In: *FM 2005: Formal Methods*. Ed. by John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 173–188. ISBN: 978-3-540-31714-2.

[12] Arkady Bron et al. “Applications of Synchronization Coverage”. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’05. ACM, 2005, pp. 206–212. ISBN: 1-59593-080-9. DOI: 10.1145/1065944.1065972. URL: http://doi.acm.org/10.1145/1065944.1065972.

[13] Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Julliand. “Scenario-based testing from UML/OCL behavioral models”. In: *International Journal on Software Tools for Technology Transfer* 13.5 (Oct. 2011), pp. 431–448. ISSN: 1433-2787. DOI: 10.1007/s10009-011-0189-7. URL: https://doi.org/10.1007/s10009-011-0189-7.

[14] Ana Cavalcanti and Marie-Claude Gaudel. “Test selection for traces refinement”. In: *Theoretical Computer Science* 563 (2015), pp. 1–42. DOI: 10.1016/j.tcs.2014.08.012. URL: https://hal.archives-ouvertes.fr/hal-01126800.

[15] Ana R. Cavalli et al. “Hit-or-Jump: An algorithm for embedded testing with applications to IN services”. In: *Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII / PSTV XIX’99, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), October 5-8, 1999, Beijing, China*. 1999, pp. 41–56.

[16] Kai Chen et al. “Testing a Distributed System: Generating Test Sequences without Potential Controllability and Observability Problems”. In: *Fifth International Conference on Networking and the International Conference on Systems (ICN / ICONS / MCL 2006), 23-29 April 2006, Mauritius*. 2006, p. 170. DOI: 10.1109/ICNICONSMCL.2006.207. URL: https://doi.org/10.1109/ICNICONSMCL.2006.207.

[17] Yoonsik Cheon and Gary T. Leavens. “A Simple and Practical Approach to Unit Testing: The JML and JUnit Way”. In: *ECOOP 2002 — Object-Oriented Programming*. Ed. by Boris Magnusson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 231–255. ISBN: 978-3-540-47993-2.

[18] Valentin Chimisliu and Franz Wotawa. “Improving Test Case Generation from UML Statecharts by Using Control, Data and Communication Dependencies”. In: *Proceedings of the 2013 13th International Conference on Quality Software*. QSIC ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 125–134. ISBN: 978-0-7695-5039-8. DOI: 10.1109/QSIC.2013.48. URL: https://doi.org/10.1109/QSIC.2013.48.

[19] Tsun S. Chow. “Testing Software Design Modeled by Finite-State Machines”. In: *IEEE Trans. Software Eng.* 4.3 (1978), pp. 178–187. DOI: 10.1109/TSE.1978.231496. URL: https://doi.org/10.1109/TSE.1978.231496.

[20] Duncan Clarke et al. “STG: A Symbolic Test Generation Tool”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Joost-Pieter Katoen and Perdita Stevens. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 470–475. ISBN: 978-3-540-46002-2.

[21] George F Coulouris et al. *Distributed Systems: Concepts and Design*. Pearson, 2011.

[22] Frédéric Dadeau, Kalou Cabrera Castillos, and Jacques Julliand. “Coverage Criteria for Model-Based Testing using Property Patterns”. In: *MBT 2014, 9th Workshop on Model-Based Testing, Satellite workshop of ETAPS 2014*. Ed. by Alexander K. Petrenko and Bernd-Holger Schlingloff. Vol. 141. EPTCS, Electronic Proceedings in Theoretical Computer Science. Grenoble, France, Apr. 2014, p. 15. DOI: 10.4204/EPTCS.141.3. URL: https://hal.inria.fr/hal-01089687.

[23] Frédéric Dadeau, Kalou Cabrera Castillos, and Régis Tissot. “Scenario-Based Testing using Symbolic Animation of B Models”. In: *Software Testing, Verification and Reliability* 6.22 (Mar. 2012), pp. 407–434. DOI: 10.1002/stvr.1467. URL: https://hal.inria.fr/hal-00760020.

[24] E. Daka and G. Fraser. "A Survey on Unit Testing Practices and Problems". In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, pp. 201–211. DOI: 10.1109/ISSRE.2014.11.

[25] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.

[26] Jin Song Dong et al. "Specifying and Verifying Sensor Networks: An Experiment of Formal Methods". In: *Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. Proceedings*. 2008, pp. 318–337. DOI: 10.1007/978-3-540-88194-0_20. URL: https://doi.org/10.1007/978-3-540-88194-0_20.

[27] A. Dunkels, B. Gronvall, and T. Voigt. "Contiki - a lightweight and flexible operating system for tiny networked sensors". In: *29th Annual IEEE International Conference on Local Computer Networks*. 2004, pp. 455–462.

[28] Bruno Dutertre. "Yicesä2.2". In: *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, 2014, pp. 737–744. ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9_49. URL: https://doi.org/10.1007/978-3-319-08867-9_49.

[29] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. "Test Generation Based on Symbolic Specifications". In: *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*. 2004, pp. 1–15. DOI: 10.1007/978-3-540-31848-4_1. URL: https://doi.org/10.1007/978-3-540-31848-4_1.

[30] Susumu Fujiwara and Gregor von Bochmann. "Testing Non-Deterministic State Machines with Fault Coverage". In: *Protocol Test Systems, IV, Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems, Leidschendam, The Netherlands, 15-17 October, 1991*. 1991, pp. 267–280.

[31] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. "From LOTOS to LNT". In: *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*. 2017, pp. 3–26. DOI: 10.1007/978-3-319-68270-9_1. URL: https://doi.org/10.1007/978-3-319-68270-9_1.

[32] Christophe Gaston, Robert M. Hierons, and Pascale Le Gall. "An Implementation Relation and Test Framework for Timed Distributed Systems". In: *Testing Software and Systems - 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings*. 2013, pp. 82–97. DOI: 10.1007/978-3-642-41707-8_6. URL: https://doi.org/10.1007/978-3-642-41707-8_6.

[33] Christophe Gaston et al. "Symbolic Execution Techniques for Test Purpose Definition". In: *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006, New York, NY, USA, May 16-18, 2006, Proceedings*. 2006, pp. 1–18. DOI: 10.1007/11754008_1. URL: https://doi.org/10.1007/11754008_1.

[34] Marcus Gerhold and Mariëlle Stoelinga. "Model-Based Testing of Probabilistic Systems". In: *Fundamental Approaches to Software Engineering*. Ed. by Perdita Stevens and Andrzej Wąsowski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 251–268. ISBN: 978-3-662-49665-7.

[35] Andreas Griesmayer et al. "Dynamic Symbolic Execution for Testing Distributed Objects". In: *Tests and Proofs*. Ed. by Catherine Dubois. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 105–120. ISBN: 978-3-642-02949-3.

[36] Anders Hessel et al. "Testing Real-Time Systems Using UPPAAL". In: *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*. 2008, pp. 77–117. DOI: 10.1007/978-3-540-78917-8_3. URL: https://doi.org/10.1007/978-3-540-78917-8_3.

[37] Anders Hessel et al. “Time-Optimal Real-Time Test Case Generation Using Uppaal”. In: *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*. 2003, pp. 114–130. DOI: 10.1007/978-3-540-24617-6_9. URL: https://doi.org/10.1007/978-3-540-24617-6_9.

[38] J. Hui and R. Kelsey. *Multicast Protocol for Low-Power and Lossy Networks, Request for Comments: 7731*. Tech. rep. Silicon Labs, February 2016.

[39] Eisuke Itoh et al. “Ordered sequence testing criteria for concurrent programs and the support tool”. In: *Proceedings of 1st Asia-Pacific Software Engineering Conference* (1994), pp. 236–245.

[40] ITU. *Z.100 : Specification and Description Language - Overview of SDL-2010*. INTERNATIONAL TELECOMMUNICATION UNION, 2010. URL: https://www.itu.int/rec/T-REC-Z.100.

[41] Claude Jard and Thierry Jéron. “TGV: theory, principles and algorithms”. In: *STTT* 7.4 (2005), pp. 297–315.

[42] Thierry Jéron. “Symbolic Model-based Test Selection”. In: *Electronic Notes in Theoretical Computer Science 240* (2009).

[43] Sen K. and Agha G. “CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools”. In: *Computer Aided Verification, 18th International Conference, CAV*. 2006, pp. 419–423.

[44] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

[45] Moez Krichen. “A Formal Framework for Conformance Testing of Distributed Real-Time Systems”. In: *Principles of Distributed Systems*. Ed. by Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 139–142. ISBN: 978-3-642-17653-1.

[46] Moez Krichen and Stavros Tripakis. “Black-Box Conformance Testing for Real-Time Systems”. In: *SPIN* (2004.).

[47] S. Kurt et al. “Packet Size Optimization in Wireless Sensor Networks for Smart Grid Applications”. In: *IEEE Trans. Industrial Electronics* 64.3 (2017), pp. 2392–2401.

[48] Frantzen L., Tretmans J., and Willemse T. A. C. “A Symbolic Framework for Model-Based Testing”. In: *Int. Workshops FATES and RV*. 2006.

[49] O. Rafiq L. Cacciari. “Controllability and observability in distributed testing”. In: *Information and Software Technology* (1999).

[50] E. A. Lee and D. G. Messerschmitt. “Synchronous data flow”. In: *Proceedings of the IEEE* 75 (1987), pp. 1235–1245.

[51] P. Levis et al. *The Trickle Algorithm, Request for Comments: 6206*. Tech. rep. March 2011.

[52] P. Levis et al. “TinyOS: An Operating System for Sensor Networks”. In: *Ambient Intelligence*. Ed. by Werner Weber, Jan M. Rabaey, and Emile Aarts. Springer Berlin Heidelberg, 2005, pp. 115–148.

[53] P. Levis et al. “Trickle: A Self-regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks”. In: *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*. NSDI’04. Berkeley, CA, USA: USENIX Association, 2004.

[54] Philip Levis et al. “The Emergence of a Networking Primitive in Wireless Sensor Networks”. In: *Communications of the ACM, Volume 51, Issue 7* (2008).

[55] Bruno Lima and Joao Pascoal Faria. “A Survey on Testing Distributed and Heterogeneous Systems: The State of the Practice”. In: *Software Technologies, 11th International Joint Conference, ICSOFT 2016, Lisbon,*

Portugal, July 24-26, 2016, Revised Selected Papers. 2016, pp. 88–107. DOI: 10.1007/978-3-319-62569-0_5. URL: https://doi.org/10.1007/978-3-319-62569-0_5.

[56] David Lugato, Nicolas Rapin, and Jean-Pierre Gallois. "Verification and tests generation for SDL industrial specifications with the AGATHA". In: 2001.

[57] Lina Marsso, Radu Mateescu, and Wendelin Serwe. "TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. Cham: Springer International Publishing, 2018, pp. 211–228. ISBN: 978-3-319-89963-3.

[58] T. Meyfroyt et al. "On the scalability and message count of Trickle-based broadcasting schemes". In: *Queueing Syst.* 81.2-3 (2015), pp. 203–230.

[59] Robert M.Hierons. "Generating Complete Controllable Test Suites for Distributed Testing". In: *IEEE Transactions on Software Engineering* (March 2015).

[60] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. USA: John Wiley & Sons, Inc., 2004. ISBN: 0471469122.

[61] Ngo Minh Thang Nguyen et al. "Behavioral Models and Scenario Selection for Testing IoT Trickle-Based Lossy Multicast Networks". In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*. IEEE, 2019, pp. 168–175.

[62] Soria Dustmann O., Sasnauskas R., and Wehrle K. "Symbolic System Time in Distributed Systems Testing". In: *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST*. 2012, pp. 893–894.

[63] OMG. *OMG Systems Modeling Language (OMG SysML), Version 1.3*. Object Management Group, 2012. URL: http://www.omg.org/spec/SysML/1.3/.

[64] Sasnauskas R. et al. "Integration testing of protocol implementations using symbolic distributed execution". In: *20th IEEE International Conference on Network Protocols, ICNP*. 2012, pp. 1–6.

[65] Sasnauskas R. et al. "Scalable Symbolic Execution of Distributed Systems". In: *2011 International Conference on Distributed Computing Systems, ICDCS*. 2011, pp. 333–342.

[66] David L. Dill Rajeev Alur. "A theory of timed automata". In: *Journal Theoretical Computer Science, Volume 126 Issue 2, April 25, 1994, Pages 183 - 235* (1994).

[67] Muhammad Jaffar-Ur Rehman et al. "Testing software components for integration: a survey of issues and techniques". In: *Softw. Test., Verif. Reliab.* 17 (2007), pp. 95–133.

[68] C. Robinson-Mallett, R. M. Hierons, and P. Liggesmeyer. "Achieving communication coverage in testing". In: *ACM SIGSOFT Software Engineering Notes* (2006).

[69] C. Robinson-Mallett et al. "Using communication coverage criteria and partial model generation to assist software integration testing". In: *Software Quality Journal* (2008).

[70] Antoine Rollet and Sebastien Salva. "Testing robustness of communicating systems using ioco-based approach". In: *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on* (2009).

[71] Vlad Rusu, Hervé Marchand, and Thierry Jéron. "Automatic Verification and Conformance Testing for Validating Safety Properties of Reactive Systems". In: *International Symposium on Formal Methods*. 2005.

[72] Khurshid S., S. Pasareanu C., and Visser W. "Generalized Symbolic Execution for Model Checking and Testing". In: *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS Proceedings*. 2003, pp. 553–568. DOI: 10.1007/3-540-36577-X_40. URL: https://doi.org/10.1007/3-540-36577-X_40.

[73] Julien Schmaltz and Jan Tretmans. “On Conformance Testing for Timed Systems”. In: *Formal Modeling and Analysis of Timed Systems* (2008).

[74] Julien Schmaltz Sabrina von Styp Henrik Bohnenkamp. “A Conformance Testing Relation for Symbolic Timed Automata”. In: *Formal Modeling and Analysis of Timed Systems* (2010).

[75] János Zoltán Szabó and Tibor Csöndes. “TITAN, TTCN-3 test execution environment”. In: 2007.

[76] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. 2007.

[77] Jan Tretmans. “Conformance Testing with Labelled Transition Systems : Implementation Relations and Test Generation”. In: *Computer Networks and ISDN Systems* (1996).

[78] Jan Tretmans. “Test generation with inputs, outputs and repetitive quiescence”. In: *Software—Concepts and Tools:103-120* (1996).

[79] Jan Tretmans and Ed Brinksma. “TorX: Automated Model Based Testing - Côte de Resyste”. In: 2003.

[80] P. Tripathy and K Naik. “Generation of Adaptive Test Cases From Non-deterministic Finite State Models”. In: (1992), pp. 309–320.

[81] Edward P. K. Tsang. “Foundations of constraint satisfaction”. In: *Computation in cognitive science*. 1993.

[82] Andreas Ulrich, Peter Zimmerer, and Gunther Chrobok-Diening. “Test Architectures for Testing Distributed Systems”. In: 1999.

[83] Juri Vain et al. “Automatic Distribution of Local Testers for Testing Distributed Systems”. In: *Databases and Information Systems IX - Selected Papers from the Twelfth International Baltic Conference, DBIS 2016, Riga, Latvia, July 4-6, 2016*. 2016, pp. 297–310.

[84] Xiao Wang and Jin Hua Zhang. “Analysis and Research on Distributed Network Protocol Testing Controllability Problem”. In: *International Conference on Information Engineering for Mechanics and Materials* (2015).

[85] Colin Willcock et al. *An Introduction to TTCN-3*. 2nd. Wiley Publishing, 2011. ISBN: 0470663065, 9780470663066.

[86] T. Winter et al. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, Request for Comments: 6550*. Tech. rep. Cooper Power Systems, Cisco Systems, and Stanford University, March 2012.

[87] M. Woehrle, R. Bakhshi, and M. Mousavi. “Mechanized Extraction of Topology Anti-patterns in Wireless Networks”. In: *Integrated Formal Methods*. Ed. by John Derrick et al. Springe, 2012, pp. 158–173.

[88] M. Zheng et al. “Towards a Model Checker for NesC and Wireless Sensor Networks”. In: *Formal Methods and Software Engineering*. Ed. by Shengchao Qin and Zongyan Qiu. Springer, 2011, pp. 372–387.

université PARIS-SACLAY

ÉCOLE DOCTORALE
INTERFACES
Approches Interdisciplinaires : Fondements, Applications et Innovation

Titre: Génération de cas de test pour les modèles symboliques de système distribués: Application au protocole IoT à base de Trickle

Mots clés: Systèmes distribués, Test à base de modèles, Systèmes de Transition Symbolique Temporisé avec des Entrées et Sorties, Génération de cas de test, Critères de couverture, Protocole MPL

Résumé: Les systèmes distribués sont composés de nombreux sous-systèmes distants les uns des autres. Afin de réaliser une même tâche, les sous-systèmes communiquent à la fois avec l'environnement par des messages externes et avec d'autres sous-systèmes par des messages internes, via un réseau de communication. En pratique, les systèmes distribués mettent en jeu plusieurs types d'erreurs, propres aux sous-systèmes les constituant, ou en lien avec les communications internes. Afin de s'assurer de leur bon fonctionnement, savoir tester de tels systèmes est essentiel. Cependant, il est très compliqué de les tester car sans horloge globale, les sous-systèmes ne peuvent pas facilement synchroniser leurs envois de messages, ce qui explique l'existence des situations non déterministes.

Le test à base de modèles (MBT) est une approche qui consiste à vérifier si le comportement d'un système sous test (SUT) est conforme à son modèle, qui spécifie les comportements souhaités. MBT comprend deux étapes principales: la génération de cas de test et le calcul de verdict. Dans cette thèse, nous nous intéressons à la génération de cas de test dans les systèmes distribués.

Nous utilisons les systèmes de transition symbolique temporisé à entrées et sorties (TIOSTS) et les analysons à l'aide des techniques d'exécution symbolique pour obtenir les comportements symboliques du système distribué. Dans notre approche, l'architecture de test permet d'observer au niveau de chaque sous-système à la fois les messages externes émis vers l'environnement et les messages internes reçus et envoyés. Notre framework de test comprend plusieurs étapes: sélectionner un objectif de test global, défini comme un comportement particulier exhibé par exécution symbolique, projeter l'objectif de test global sur chaque sous-système pour obtenir des objectifs de test locaux, dériver des cas de test unitaires pour chacun des sous-systèmes. L'exécution du test consiste à exécuter des cas de test locaux sur les sous-systèmes paramétrés par les objectifs de tests en calculant à la volée les données de test à soumettre au sous-système en fonction de données observées. Enfin, nous mettons en œuvre notre approche sur un cas d'étude décrivant un protocole utilisé dans le contexte de l'IoT.

Title: Test case generation for Symbolic Distributed System Models: Application to Trickle based IoT Protocol

Keywords: Distributed systems, Model-Based Testing, Timed Input Output Symbolic Transition Systems, Test case generation, Coverage criteria, MPL protocol

Abstract: Distributed systems are composed of many distant subsystems. In order to achieve a common task, subsystems communicate both with the local environment by external messages and with other subsystems by internal messages through a communication network. In practice, distributed systems are likely to reveal many kinds of errors, so that we need to test them before reaching a certain level of confidence in them. However, testing distributed systems is complicated due to their intrinsic characteristics. Without global clocks, subsystems cannot synchronize messages, leading to non-deterministic situations.

Model-Based Testing (MBT) aims at checking whether the behavior of a system under test (SUT) is consistent with its model, specifying expected behaviors. MBT is useful for two main steps: test case generation and verdict computation. In this thesis, we are mainly interested in the generation of test cases for distributed systems.

To specify the desired behaviors, we use Timed Input Output Symbolic Transition Systems (TIOSTS), provided with symbolic execution techniques to derive behaviors of the distributed system. Moreover, we assume that in addition to external messages, a local test case observes internal messages received and sent by the co-localized subsystem. Our testing framework includes several steps: selecting a global test purpose using symbolic execution on the global system, projecting the global test purpose to obtain a local test purpose per subsystem, deriving unitary test case per subsystem. Then, test execution consists of executing local test cases by submitting data compatible following a local test purpose and computing a test verdict on the fly. Finally, we apply our testing framework to a case study issued from a protocol popular in the context of IoT.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France