Model-Based Testing Real-Time and Interactive Music Systems

Clément Poncelet Sanchez

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Clément PONCELET SANCHEZ

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Model-Based Testing Real-Time and Interactive Music Systems

soutenue le 10 novembre 2016

devant le jury composé de :

Florent Jacquemard	Directeur de thèse
Camilo Rueda	Rapporteur
Mariëlle Stoelinga	Rapporteuse
Carlos Agon	Examinateur
Marc Aiguier	Examinateur
Alexandre Donze	Examinateur
Emmanuelle Encrenaz	Invitée
Jean-Louis Giavitto	Examinateur
David Janin	Examinateur
Didier Lime	Examinateur

Abstract

Can real-time interactive systems be automatically timed tested ? This work proposes an answer to this question by providing a formal model based testing framework for Interactive Music Systems (IMS).

IMSs should musically perform computations during live performances, accompanying and acting like real musicians. They can be *score-based*, and in this case must follow at all cost the timed high-level requirement given beforehand, called *score*. During performance, the system must react in real-time to audio signals from musicians according to this score. Such goals imply strong needs of temporal reliability and robustness to unforeseen errors in input. Be able to formally check this robustness before execution is a problem insufficiently addressed by the computer music community.

We present, in this document, the concrete application of a Model-Based Testing (MBT) framework to a state-of-the-art IMS. The framework was defined on purpose of testing real-time interactive systems in general.

We formally define the model in which our method is based. This model is automatically constructed from the high-level requirements and can be translated into a network of time automata. The mixed music environment implies the management of a multi-timed context and the generation of musically relevant input data through the testing framework. Therefore, this framework is both time-based, permitting durations related to different time units, and event-driven, following the musician events given in input.

In order to test the IMS against the user's requirements, multiple options are provided by our framework. Among these options, two approaches, *offline* and *online*, are possible to assess the system timed conformance fully automatically, from the requirement to the verdict. The *offline* approach, using the model-checker Uppaal, can generate a covering input suite and guarantee the system time reliability, or only check its behavior for a specific or fuzzed input sequence. The *online* approach, directly interprets the model as byte-code instructions thanks to a virtual machine. Finally, we perform experiments on a real-case study: the score follower Antescofo. These experiments test the system with a benchmark of scores and a real mixed-score given as input requirements in our framework. The results permit to compare the different options and scenarios in order to evaluate the framework.

The application of our fully automatic framework to real mixed scores used in concerts have permitted to identify bugs in the target IMS.

Résumé

Est-il possible de tester automatiquement le comportement temporisé des systèmes interactifs temps réel ? Ces travaux proposent une solution en fournissant un *ensemble d'outils* de test basé sur modèles pour Systèmes Musicaux Interactifs (SMI).

Les SMIs doivent calculer et réagir pendant une performance musicale et ainsi accompagner les musiciens. Certains de ces SMIs peuvent être basés sur partition et doivent, dans ce cas, suivre à tout prix les contraintes temporelles imposées par le document haut-niveau appelé partition. En somme, pendant une performance, le système doit réagir en temps réel aux signaux audio venant des musiciens en suivant cette partition. Ceci demande au système une forte fiabilité temporelle et une robustesse face aux erreurs pouvant arriver en entrée du système. Hors, la vérification formelle de propriétés, comme la fiabilité temporelle avant l'exécution du système lors d'une performance, est insuffisamment traitée par la communauté de l'informatique musicale.

Nous présentons dans cette thèse, la réalisation d'un *ensemble d'outils* de test basé sur modèles appliqué à un SMI. Il est à noter que ces *outils* de test ont été définis formellement dans le but de tester plus généralement le comportement temporelle des systèmes interactifs temps réel prenant en compte des évènements discrets et des durées définissables sur des échelles multiples.

Pour ce résumé nous présentons rapidement l'état de l'art de nos travaux avant d'introduire la définition de notre modèle créé pour spécifier les aspects évènementiel («event-triggerred») et temporel («timed-driven») des SMIs. Ce modèle a la particularité d'être automatiquement construit depuis les conditions temporelles définies dans un document haut-niveau et peut être traduit vers un réseau d'Automates Temporisés (TA). Dans le cadre de la performance musique mixte électronique/instrumentale nous avons introduit une notion de durée multi-temps gérée par notre modèle et une génération de trace d'entrée musicalement pertinente par notre *ensemble d'outils* de test.

Pour tester un SMI selon les différentes attentes de l'utilisateur, notre *ensemble d'outils* a été implémenté avec plusieurs options possibles. Parmi ces options, la possibilité de tester automatiquement, selon une approche différée ou temps réel, la conformité temporelle du SMI est proposée. En effet, l'approche différée utilise des outils de la gamme du logiciel Uppaal [44] pour générer une suite de traces d'entrées exhaustive et garantir la conformité temporelle du système testé. Il est également possible de tester

une trace d'entrée particulière ou une version altérée («fuzzed») de la trace idéale définie par la partition. L'approche temps réel interprète quand-à elle directement le modèle comme des instructions de byte-code grâce à une machine virtuelle. Finalement, des expériences ont été conduites via une étude de cas sur le suiveur de partition Antescofo.

Ces expériences ont permis de tester ce système et d'évaluer notre *ensemble d'outils* et ses différentes options. Ce cas d'étude applique nos *outils* de test sur Antescofo avec succès et a permit d'identifier des bogues parfois non triviaux dans ce SMI.

Acknowledgements

I wish to thank every smile, salutation, talk and discussion during these three years. I especially thank the amazing laboratory' members, my helpful supervisor and the unbelievable mates I could meet in my office.

Summary

Chapter 1

Introduction

Mixed music is concerned with musical pieces involving human and electronic devices, playing together. The music workflow is generally divided in two distinct steps - composition, where pieces are thought and scores written; and performance, where musicians interpret a score and shows take place. Following this division, mixed music and even computer music systems fall into two categories, usually distinct:

1) Music authoring systems, used to write a score and/or compose a music piece, and

2) Real-time performance systems, for live execution of a piece, interacting with musicians.

The formers require representations for providing a sufficient expressivity to composers, borrowing tools on - formal languages, constraint programming, visual programming languages - from the literature to deal with such a problem. The latter systems are real-time and must be efficient, they use lower level data and fast algorithms to tackle their challenge.

As a consequence, the use of formal methods (in particular static procedures) is far more developed for the authoring than for real-time systems. However, these second systems are used in concerts and a good reliability is required to ensure the system expected behaviors for any sequence of inputs (in particular with musicians interpretations and errors). Moreover, the real-time performance systems assessment is usually manual, an implementation of the system is checked during a rehearsal by an auditive mean and on few inputs performed by musicians. It is not accurate and cannot guarantee a good behavior of the system at show time for every musician's performance.

In this work, we consider score-based Interactive Music Systems [81, 19] (IMS), testing the system Antescofo in a case study. Such systems work with a *mixed score*, written in the IMS's Domain Specific Language (DSL), which describes the input expected from human musicians, together with the electronic output to be played in response. During a performance, a score-based IMS aligns in real-time the position of the human musicians to the score, handling possible errors, detects the current tempo, and plays the electronic part. It is therefore a *reactive* system, interacting with the musicians under strong timing constraints: its output (generally messages passed to an external audio application) must indeed be emitted at *the right moment*, not too late but also not too early. Thus, the development and use of such systems involve covering both the authoring and the real-time problems.

There are two well known techniques to prevent a system from raising errors [83, 32]. On the one hand, verification methods use formal tools for system developments to prove that the system has no error. On the other hand, testing techniques check already running systems in order to detect and correct errors. Usually, the first methods require expertise to express the expected requirements, but the second ones cannot guarantee a system totally without errors since they verify errors only for a tested input sequences.

Here, we use the Model-Based Testing [63] (MBT) technique, which provides formal methods for testing a system. MBT techniques require a system specification called a *model*, in which the expected behaviors of the system are specified via an abstraction which focuses only on one or few system aspects (time, input/output, communication ...). The goal of such techniques is to assess the conformance of the system implementation we want to test, called the Implementation Under Test (IUT), according to its model. A system conforms its model, if for all input sequences, the IUT reacts as the model.

Considering the spread of human-interactive systems in the fields of music, internet of things, embedded systems... and the increasing importance of such systems, the thesis motivation is to assess automatically the time conformance of a system according to a specification described as a or several time - interactive scenario(s). More precisely, the thesis objective is to investigate if formal methods can be used to test embedded systems and explore their application to IMS. Here, we consider embedded systems as reactive systems which have to send actions at or after a given input.

The first goal is to contribute in bridging the gap between authoring and real-time systems in the context of IMS by providing techniques for improving the assessment of IMSs. Moreover, MBT assistance for score authoring allows static analysis of the IMS's realtime behaviors. A second goal is to use the mixed score in order to automatically construct the IMS models, generate relevant input sequences, compute the model expected behaviors and finally compare with the implementation reactions to assess its conformance.

To this aim, a test framework was implemented and is overviewed in Figure 1.1. Briefly, two approaches have been implemented in the framework and depicted on the left and right sides of the figure. Both of these testing approaches start from an Antescofo mixed-score (on the top) and construct automatically the corresponding model (1):

- Then, on the right, the approach (called offline) generates a set of tests from the model and/or the mixed-score information (2). Once these tests have been generated, the model is used to compute the corresponding reference traces by simulation (3). Then, the same tests are sent to Antescofo (4) in order to deduce the monitored traces. Finally, the reference traces are compared to the monitored ones resulting in a verdict (5), assessing whether the tests pass.

- On the left, the second approach (called online) uses a Virtual Machine (VM) to execute the model. The test data is generated on the fly using

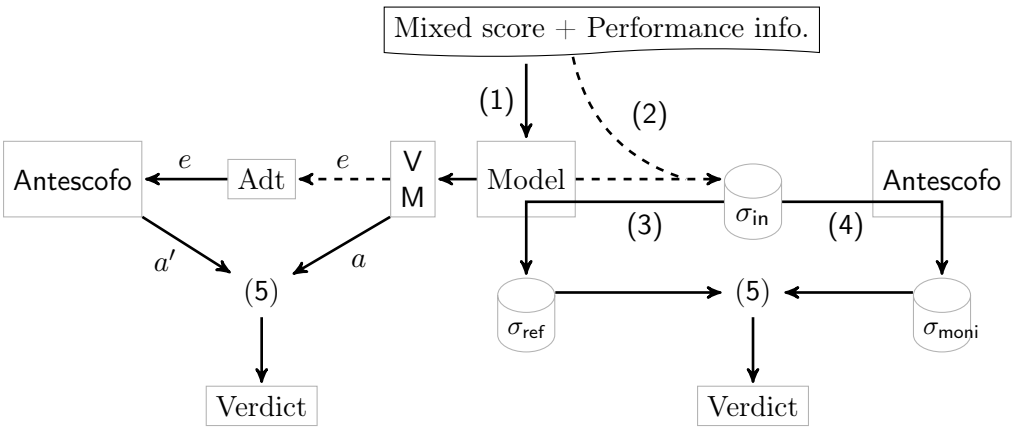

Figure 1.1: Two implementations of score-based IMS testing procedures: on the left online method - on the right offline method.

an adapter (*Adt*) and is sent to Antescofo, as the input of artificial musicians. The monitored outputs of the system are compared online to the reference trace (5), event by event. An error is reported if some Antescofo reactions are not expected or missed in respect to the model.

The contributions of the thesis are:

* A formal definition of a model, called Interactive Real-Time Model (IRTM) and discussed in Chapter 3, for specifying systems both timed-triggered with possibly multiple time scales and event-driven, with a sound translation into Time Automata model.
* A testing framework based on IRTM (Chapter 4) and implemented to test real-time systems. Two approaches are managed by the framework, in particular an offline approach involving systems from the Uppaal tool suite.
* A virtual machine (described in Section 3.3) and a related online approach for generating the input sequence on the fly while the test of a real-time system is running.
* A framework documentation and a data-set of regression testing for Antescofo.

The MBT framework for score-based IMS was published in [76, 58]. The formal definition of IRTMs and the MBT framework were published in [74]. In [75], we extended the last paper with the automatic model construction and the online MBT approach.

This thesis was co-founded by the *French Government Defense* (DGA) and the *French Institute for Research in Computer Science and Automation* (INRIA), it was a part of the UMR 9912 *Music and sound Sciences and technologies* (SMTS) and was part of the INRIA MuTant team in the RepMus team of the *Institute for Research and Coordination in Acoustics and Music* (IRCAM). The thesis doctoral school was EDITE hosted by the *University Pierre and Marie Curie* (UPMC) Sorbonne University.

The document is organized as follows: First, Chapter 2 presents the state of the art, detailing the mixed score context (Section 2.1) and existing MBT formal definitions (Section 2.2) used along the document. Thereafter, three chapters present our main contributions:

- Our Interactive Real-Time Model (IRTM) is formally introduced in Chapter 3. IRTMs specify the real-time systems we want to consider. The IRTM principles, syntax and standard semantics are defined in Section 3.1. In order to use existing test frameworks, a translation from

IRTM into equivalent models is formally defined and its soundness is proved in Section 3.2. Finally, a virtual machine executing such IRTMs is presented in Section 3.3.

- Our testing framework based on IRTMs is defined in Chapter 4 and follows the MBT workflow. The framework implementation for an abstract real-time system is presented in Section 4.1. Contrary to usual MBT frameworks, our procedure constructs automatically IRTMs from a requirement using construction rules presented in Section 4.2. Finally, we focus on the generation of a set of input traces in Section 4.3.

- Thereafter, a case study details the testing framework application to the score-based IMS Antescofo in Chapter 5. The IUT is presented in Section 5.1, with its specification procedure (Section 5.2). Finally, Section 5.3 reports the results computed from two experiments: a benchmark of scores and real score cases.

Chapter 2

Testing Mixed Music Systems: State of the Art

We dedicate this section to present the context and techniques related to our work. Section 2.1 introduces the mixed music and its specific time. We briefly list music systems before presenting Antescofo, the system under test in our case study. Then, some existing tools for assessing such music systems are presented. Thereafter, Section 2.2 defines the model-based testing formalization used through the document. We give our definition of model and present existing methods and tools to perform model-based testing for real-time systems.

2.1 Testing Mixed Music Systems

Music has always been an art of creativity in which sounds are produced in time by one or several musicians. The music pieces complexity has not ceased to increase, bringing through the history solid and expressive notations. These notations, as the western notation for the occidental culture, established the sustainable power of the composer creations. With the introduction of mixed music, involving electronic actions within musical pieces, a new step forward in musical expressivity was required. As a solution, the mixed score was created. It is an extension of the score document in which, aside from the musician input events, a set of output electronic actions is defined.

In this work, a mixed score is a composer requirement containing: - one or multiple sequences of ideal musician events and, sequences of actions timely related to these events -. Moreover, a musician play (called a *performance*) is considered as a sequence of event onsets (when the musician begins to play effectively an event) and a duration before the next event onset. Notice that rests are considered as event durations here.

In the remaining of this section, we detail some music notions focussing on score and performance. They both contribute substantially to the realization of a music piece and come from two main steps of a music creation process: composition and performance.

2.1.1 Composition and Authoring Systems

During composition, composers invoke creative ideas and use complex tools in order to make a piece. A myriad of documents are written, explaining the piece to be playable following the creator intentions. The mixed score is one of these documents and usually contains a specific unit for expressing its time dimension, the relative time unit. Relative time unit is an abstract time unit manipulated by composers in order to englobe every musician performance of its piece. A duration in such a time unit is measured in *beat* and is relative to the musician pace (called *tempo*) during performance. It is an important musical dimension since relative time provides a freedom of interpretation for musicians.

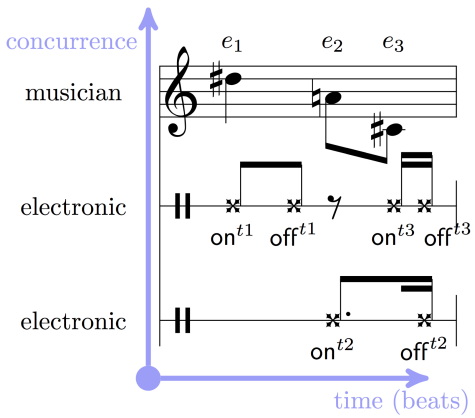

Figure 2.1: A mixed score specifying one musician part and two electronic parts, these parts are called *staves*. Each stave is a coherent phrase of notes through musical time (horizontal axis). The mixed score represents the ideal sequence of these staves, played "in concurrence", *i.e.* altogether during the performance.

Example 2.1.1: In Figure 2.1, three musical parts are specified. These parts, called *staves*, represent the ideal sequence in time (the horizontal axis) of a piece. For instance, the musician stave (on the top) specifies one note lasting one beat (called a *quarter*) with the pitch $D5^{\#}$ and a label e_1 (depicted above the note). The two next events last a half of e_1 duration (0.5 beat, called *eigth*) and have the pitch $A4$ and $C4^{\#}$ for respectively e_2 and e_3. Notice that we deal with note pitches using american pitch notation. This information is sufficient for musicians to play the stave (remark that usually the labels are not specified).

However, there are implicit and *strongly timed* synchronizations according to the vertical axis, since vertically aligned notes must be "played at the same moment". For example, the mixed score above specifies a simultaneous play of e_1 (by the musician) and on^{t1} (by the first electronic system). $\Diamond$

Authoring systems aim at simplifying the composer creation, by focusing on offline and static solutions. The literature provides a number of visual programming languages, Domain Specific Languages (DSL) or abstract representation tools in order to bring to composers a manner of expressing their imagination. As an example, we only cite a few of the existing authoring systems: OpenMusic [27], PatchWorks [67] and Common Music [84], all based on the programming language Common Lisp [86].

OpenMusic is a visual programming language allowing an utilization of symbolic music representations. The system manipulates relative time (in beat) which is not continuous as the physical time and allows the applications of generic algorithms abstracting every possible performance. Currently, in order to provide more dynamism to the authoring system, recent works added a possibility to play a performance in OpenMusic [28]. These studies tackle the lack of dynamism of such authoring systems and increase composer possibilities by providing interactions with musicians.

2.1.2 Interactive Music Systems for Performance

During performance, musicians interpret a piece by altering the ideal event durations written in the score. This interpretation decorates the piece with performers' emotions/intentions and translates the abstract time units (in beat) of the score into physical ones (in seconds). The real-time context of a performance makes it non-reproducible and unpredictable, two highly risky properties for computer systems.

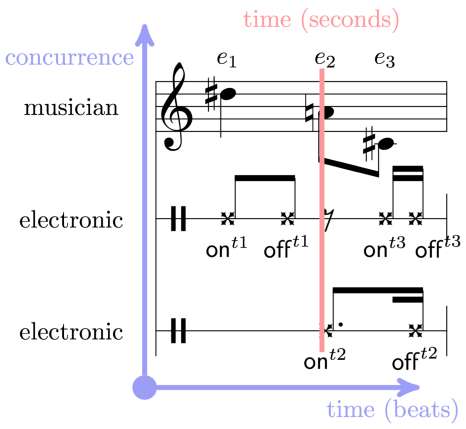

Figure 2.2: The vertical line depicts the positions of each musician or electronic system on its stave. Here, the musician is going to play e_2, the first electronic system should idle (rest) and the second should play on^{t2}. Also, the line's left side depicts the past (e_1 for the musician) and the right side the performance continuation (e_3).

Example 2.1.2: The Figure 2.2 depicts a performance using a vertical line on the score. The line symbolizes the current position of the musicians/systems on the staves. Hence, it implies that the notes before the line have been played and those after should be played. We define a sequence of played events by the dates when their onset were played relatively to the beginning of the performance. As if a timer was launched at the first onset and for each next onset its timer value is stamped onto them. For example, a musician can play e_1 at 0 second, e_2 at 1 second and e_3 at 1.25 seconds, we call these values *timestamps*.

Each musician has a *tempo* in beat per minute (bpm) when playing a performance which can be computed thanks to the translation function. Indeed, 1 beat lasts 1 second with a tempo of 60_{bpm}, 2 seconds with 30_{bpm} and 0.50 second with 120_{bpm}. After computing the event durations using the next event timestamp, it is so possible to deduce the musician pace. In our case, e_1 lasted 1 second and e_2 0.25 second. Hence, e_1 was played with 60_{bpm} and e_2 with 120_{bpm} because it lasted 0.25 second for 0.5 beat in the score. However in real cases, one different tempo is viewed for each musician, therefore, the concrete position of each one is not aligned in a vertical line with the other positions. ◊

Interactive Music Systems (IMS) aim at being involved in concerts. In [81] and [19] a generic definition of IMS is presented. We depict this definition in Figure 2.3. IMSs are systems interacting in real-time with performers (musicians, dancers ...) and audience (concert listeners, piece watchers ...). In order to interact with its environment, such systems use sensors and actuators, a sensor detection inducing the system reaction. In this thesis, we restrict this definition to a music application and define the IMS as an

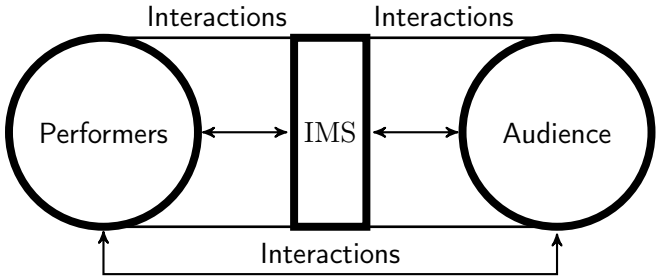

Figure 2.3: Interactive Music Systems in concerts, communicating via sensors/actuators (based on Bert Bonger [19] definition).

electronic musician mimicking human ones. It implies several actions:

* input detections,
* real-time reactions to inputs, and
* anticipation of future actions.

IMSs detect inputs by listening to or tracking of performer(s). These systems are event triggered and must react instantaneously at an input detection. Finally, they manage timed synchronizations or/and compute the musician's pace for anticipating and adapting their reactions accordingly. In practice, pedals are used to synchronize with musicians (and compute easily their pace) and stochastic models estimate future durations for anticipating in real-time. Usually in reactions, IMSs can preform sound processing or message sending to other audio applications.

The most demanding problem of IMSs is the temporal reliability. Indeed they consider time as a critical resource since an output may have a precise value and a precise date (not only the "what" but the "when" is primordial). A IMS's reaction must not be undertaken too early or too late.

As an example, we cite some IMSs in the literature, Formula [4], MAX-MSP [78] and PureData [79] and Chuck [48].

MAX-MSP [78] and PureData [79] are visual programming language environments. They are long-established and provide a simple way to construct dynamic and reactive systems (called *patches*). A patch consists in plugging *functions* or *sub-patches* to each other, which are sequentially linked and executed in real-time. These visual data-flow languages are particularly effective for hierarchical control and signal processing modules creation.

Chuck [48] is a *strongly timed* programming language that provides a linker keyword called the Chuck operator => and an explicit notion of time,

mapped into a number of audio samples, which can be manipulated in the language. At runtime, a virtual machine, called Shreduler, runs all the Shreds, Chuck programs, in concurrence with the audio management (the audio Unit Generators (UGen) a synthesis network). Explicit time eases Chuck's synchronization and concurrence management, indeed, the Shreduler knows at runtime when the Shreds wake-up and stop. This feature is called the *time-mediated concurrency* which is appreciated by people enjoying *live coding*.

Jitter problem. A challenge for IMSs is to compute and produce sounds while dynamically taking into account user wishes. Indeed, when producing continuous sounds, a minimum rate (around 20 ms) of data buffers has to be provided by the system in order to avoid incoherent glitches in the output (called "clicks"). However, IMSs should sometimes compute complex algorithms for controlling and making the expected sound.

2.1.3 The score-based Interactive Music System Antescofo

Among IMSs we distinguish *score-based* IMSs, which are constrained to follow a pre-specified timed scenario during performance (generally given beforehand as input). These particular systems must tackle the *score following* problem, consisting in localizing a musician position on a given score during performance.

We consider for our work the IMS Antescofo as the system to test in our case study. Antescofo is a score-based IMS performing score following. In order to introduce the system, we highlight the historical evolutions of musical human-system interactions:

1939 The first interactions were created with the electronics part fixed on a support (*i.e.* audio records) followed by musicians at performance. For instance, "Imaginary Landscape No.1" by *John Cage* (1939), or later, the piece "Kontakte" for piano, drums and tape by *Karlheinz Stockhausen* (1958-60). In the second piece, the interactions were expressed as audio timestamps on the top of the score to synchronize and localize the musicians according to the records as shown on the left of the Figure 2.4.

1983 Score following algorithms are created to allow systems to compute the current humans positions on a score, as in "Barry Vercoe" by *Roger Dannenberg* (1983). In this piece, the system used an abstract MIDI input to follow the musician performance.

1990's Hidden Markov Models [52] (HMM) introduce probabilities in speech and sound processing. They enabled a finer and smarter score following directly from audio waves and improved online algorithms.

2008 Algorithms enabled anticipative score following which extracts online the position and the tempo of musicians. This idea is implemented in the system Antescofo and is a step closer to a machine mimicking human musician behaviors.

This evolution is depicted in Figure 2.4. It presents on the left an extract of "Kontakte" by *Karlheinz Stockhausen* (1958-60) and on the right an Antescofo mixed score.

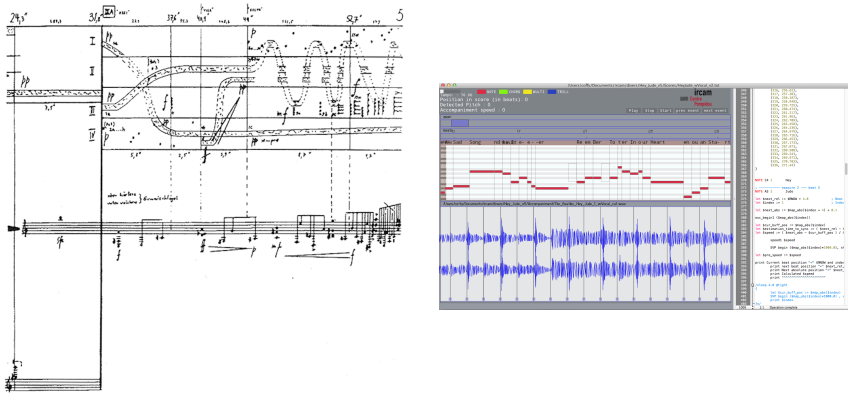

Figure 2.4: Evolution of score based IMS systems, on the left an extract of "Kontakte" by *Karlheinz Stockhausen*, on the right an example of the Antescofo visual interface Ascograph for an automatic accompaniment piece.

Antescofo was first developed as a MAX-MSP object to be embedded in a *patch* (available in PureData too). Also, during performance, Antescofo usually waits for audio signal or midi input stream and sends output *messages*. As a score based IMS system, Antescofo belongs both to the authoring and real-time IMS systems:

- Authoring: Antescofo requires a mixed-score written in a Domain Specific Language (DSL), specifying the events to detect and the electronics actions to send. The system provides authoring features to aid composers for writing an Antescofo mixed score enabling possible complex algorithms in the piece.
- Performance: After processing the mixed score, Antescofo: - waits for an event produced by humans, detects it, processes score following and

finally reacts by sending a message -. All these actions are performed online during performance.

A *standalone* version of Antescofo was implemented. The standalone should use a virtual clock to run in a fast-forward fashion (*i.e.* there is no real-time management). It means that instead of waiting for a duration in real-time, the system notifies that an amount of time is passed to its virtual clock and continues its execution.

2.1.4 Testing Interactive Music Systems

The context of music imposes us unpredictable and free input events received from the environment. Even if a musician does not make any mistakes in its play, the detection process by itself can erroneously detect wrong pitches or altered durations. Because IMSs are involved in concerts with human-musicians, they should not crash or report errors: "the show must go on!" However, how currently are IMSs tested and verified to prevent from crashes at show-time?

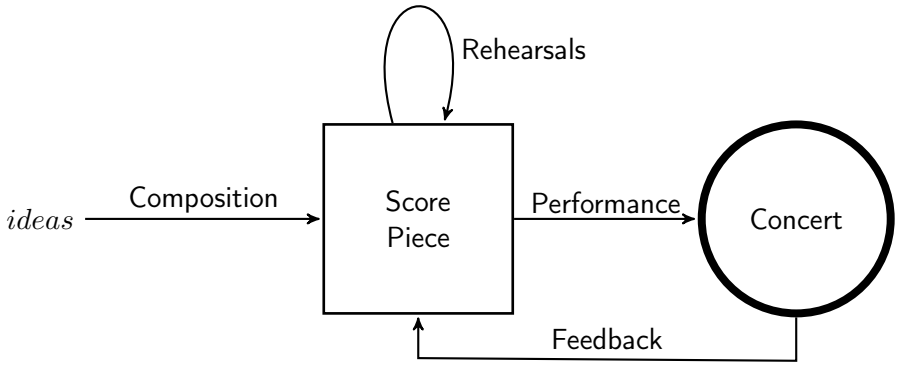

Figure 2.5: General sketch of a piece lifetime.

Rehearsals. Figure 2.5 depicts a general sketch of a piece lifetime. First, a score is created from ideas (and magics) by composers and several formats of this score are written (paper, electronics or both). Second, rehearsals, in a concert situation, test the IMS behaviors while the composers retrieve a global feedback on the piece. Then, the performance takes place and the show starts giving feedback for future performances.

Although rehearsals are effective to assess the global feeling of a piece, they are not actually dedicated to IMSs testing. Indeed, rehearsals are costly in time and money (concert simulation) and cannot be disrupted by system crashes. Moreover, multiple drawbacks can be highlighted:

- Only the performance played during the rehearsal is tested which differs from the concert one.
- It is real-time, in the sense that the piece has to be played physically, hence it takes one hour to test a single performance on a complete one hour long piece.
- This method is tedious since the composers listen to the result to assess the IMS.

Finally, the rehearsal is more an artistic judgement than a IMS debugging. For this reasons another way to find and fix IMS bugs has to be found.

Given a system to test, called *Implementation Under Test* (IUT), two well-known approaches consist in regarding or not the IUT's source code in order to test the system. *White-box* testing [88, 29] is based on traversing the source code (or its abstraction) of the IUT. On the contrary, *black-box* testing [88, 29] focuses only on the IUT inputs/outputs to test. We follow this two approaches in order to introduce and present existing tools for testing IMS. Moreover, we highlight the pros and cons for testing timed behaviors of systems and estimating the tool efficiency in our context.

White-box testing Examples of usual white-box testing are *assertions* inserted on the source code or *fuzzing* techniques which traverse instructions with a set of system inputs in order to assess absence of errors.

Assertions are code instructions and check, using predicates or boolean functions, variable values (input or output). They are useful to stop the system when an unexpected value is detected. MAX-test package in MAX-MSP [38] developed this feature for testing MAX patches. It provides an automatic tool for testing IMSs but requires both the expected and IUT's values that have to be computed in another manner. Assertion techniques are effective to test if a value is between bounds (*i.e.* for parameters) however it is related to the code and fails to provide a manner to compute precise *expected values*. Therefore, assertions cannot be used for testing temporal properties.

Fuzzing techniques [51] have the advantage to be easy to set up since its goal is to start with a single input sequence and mutate it, in general randomly, to generate a set of inputs. The suite of input sequences is then

stimulated on the IUT for monitoring its behavior. In [20], a fuzzing method is used directly in production, testing an operating system. This technique allows to test a wide number of programs with many input sequences. Moreover, they use constraint solvers and keep track of the instructions tested to mutate the best input and optimize the coverage of the input trace suite generated. However, only crashes and critical errors are checked with fuzzing techniques and, similarly to assertions, these techniques fail to provide expected values during the test. Although random generation is important for testing because it always raises unexpected errors, this generation is ineffective for covering code (*i.e.* executing all the instructions of a large code). Finally, this technique cannot test the IUT's temporal behaviors.

Black-box testing We focus the presentation of black-box testing techniques on Model-Based Testing (MBT) techniques. A model is a simple specification of the IUT, abstracting few of its aspects. Models usually are human readable graphs, in which nodes abstract IUT states and transitions IUT observable behaviors between states. The goal of a model is to give an easy specification regarding some characteristics of the system, *i.e.* its communications, its time behaviors, its input/outputs and be able to make formal reasonings.

This method is the solution we have chosen to test the IMS Antescofo.

Some works on modeling the IMS i-Score [7, 6] have been produced. They use timed automaton or petri-net models for verifying the timing and communication behaviors of the IMS. However, these models of i-Score focus on reasoning about formal model properties rather than IMS testing.

Usually, the main limitation of MBT is the manual construction of models that is tedious and error prone. However, MBT is effective to specify temporal behavior. For our work, we follow MBT techniques for testing IMS and designed an automatic solution to ease the model constructions.

2.2 Model of Timed System

Model-based testing (MBT) [63] is a formal method to test whether a system conforms a *specification*. In the remaining of the document, we call *specification* or *model* an abstract representation of a concrete system. A model allows to reason automatically on system properties and is usually a simpler description of this system. MBT uses models in order to generate a suite of tests for assessing an implementation of the concrete system, called Implementation Under Test (IUT).

In the following of the section, we present relevant models to specify real-time systems. Thereafter, the MBT method is detailed with two different approaches of implementation.

2.2.1 Input/Output System Models

A model can be defined over a set of actions A abstracting the input/output behaviors of a system. Usually, MBT methods are based on Finite State Machines (FSM), an abstract machine with finite sets of locations, edges and actions. A location abstracts a system's state and its outgoing edges the possible actions the system can do through states.

Model of Interactions Real-time systems are usually specified using *states* (abstracting a state of the concrete system), *transitions* (possible actions from a state) and three kinds of action: input, output and internal actions. These actions specify an input expectation, an output emission and an internal system computation respectively. Input-Output Labeled Transition Systems (IOLTS) [87, 85] are models with finite sets of states, transitions and actions. Each transition is labeled with an action specifying an interaction with their environment. It is a simple model allowing the specification of open systems where some inputs are expected and some outputs returned.

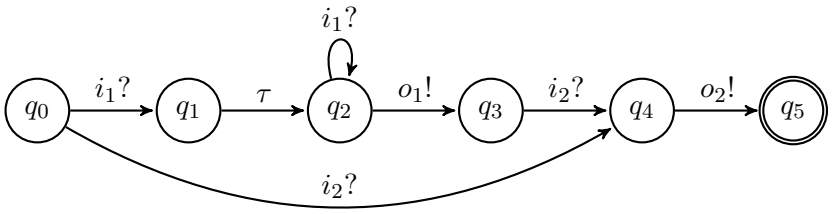

Figure 2.6: IOLTS example: Playing o_i when receiving the corresponding i_i.

Example 2.2.3: The example in Figure 2.6 depicts an IOLTS model. The model states are depicted with circles and the transitions are depicted with arrows, starting from a state called *source* and guiding to the *target* state. An action is depicted $i_i?$, for receiving an event (modeling an event detection); $o_i!$, for emitting an output; and τ, for performing an internal action. The double circled state (q_5) designs the ending state, called *exit* state, which is a state without outgoing transition.

We specified in this figure a system reacting to two inputs i_1 and i_2. The system must send the corresponding output o_i only once for $i \in \{1, 2\}$. However, if i_2 is received before i_1 then o_1 is discarded. The model abstracts an internal action τ performed before returning o_1.

It is a typical IMS behavior which waits for event detections and needs to anticipate or manage a missed event during performance. One can imagine that inputs i_1 and i_2 are notes, chords or more complex event detections from a listening machine. The internal action can abstract an effect computation from the input i_1 (for example echoing the input with a disturbance) resulting to the output o_1. o_2 can be a simpler atomic command, to switch off the output flow. Then for example, a possible model might describe that if the first input is detected as expected (in a normal case), the effect is computed for the output o_1 . Otherwise, if the musician played i_2 and missed i_1, the system has to omit this echo because the relevant input was not played, and the output o_2 is sent directly. ◊

Definition 2.1. *A IOLTS is a 4-tuple* $\langle \mathcal{Q}, q_0, \mathsf{A}, \mathrm{T} \rangle$ *such as:*

- $\mathcal{Q}$ *is a non-empty and finite set of states,*
- $q_0 \in \mathcal{Q}$ *is the unique initial state,*
- A *is a set of actions such that* $\mathsf{A} = \mathsf{A}_{\mathsf{in}} \mathbin{\mathring{\cup}} \mathsf{A}_{\mathsf{out}} \mathbin{\mathring{\cup}} \{\tau\}$*:*
  - A_{in} *is set of input actions,*
  - $\mathsf{A}_{\mathsf{out}}$ *is set of output actions,*
  - $\{\tau\}$ *is the generic internal action.*
- $\mathrm{T} \subset \mathcal{Q} \times \mathsf{A} \times \mathcal{Q}$ *is a set of transitions.*

with $\mathring{\cup}$ *the union operation on disjoint sets.*

We denote as $i? \in \mathsf{A}_{\mathsf{in}}$, the wait for an input i and $o! \in \mathsf{A}_{\mathsf{out}}$, the emission of an output o. The set $\mathsf{A}_{\mathsf{V}} = \mathsf{A}_{\mathsf{in}} \mathbin{\mathring{\cup}} \mathsf{A}_{\mathsf{out}}$ is the set of observable actions, these actions are visible from outside of the model. We commonly use μ to denote an emission or a wait for a symbol in A, and α in A_{V}.

A IOLTS is efficient in specifying input-output relations but time is not considered in the model. Therefore, an infinite amount of time can last at each state as for example between i_i and o_i in Figure 2.6.

2.2.2 Time Modeling

Timed Automaton (TA), defined in the 90's by Alur and Dill [2], is well-known for specifying time within a model. TAs are finite automata manipulating variables called *clocks*. A clock value domain is defined on the

nonnegative reals $\mathbb{R}_{\geq 0}$, in order to abstract and manipulate time. Thus, every duration or clock value is in model time unit (mtu). The principle of TAs is that all clocks advance synchronously with the same amount of time but can be independently set to 0. In the model, clocks are used through constraints in two cases: *guards*, in transitions, to forbid an execution of a transition when the constraint does not hold; *invariants*, in locations, to permit their occupation only when the constraint holds. We present Timed Automata with Input Output (TAIO) defined in [64] which is a TA with a set of actions $\mathsf{A} = \mathsf{A}_{\mathsf{in}} \mathbin{\mathring{\cup}} \mathsf{A}_{\mathsf{out}} \mathbin{\mathring{\cup}} \{\tau\}$ and *urgent* locations. An urgent location is a location in which time is frozen and cannot advance.

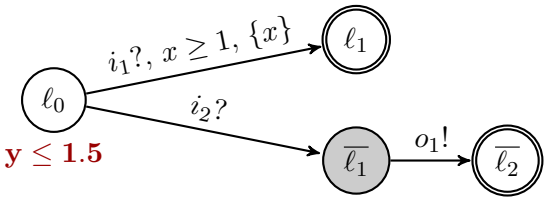

Figure 2.7: Principle of TAIO models.

Example 2.2.4: Figure 2.7 depicts a TAIO containing two clocks, x and y. Each transition restricts the clock values with a guard and resets a set of clocks when fired. For example the transition ℓ_0 to ℓ_1 can be taken only if $x \geq 1$ and resets x to 0, if not depicted the guard is true and the set is empty. The duration lasted at a location can be restricted by an invariant, depicted below a location (*e.g.* ℓ_0). Grey locations (such as $\overline{\ell_1}$) are *urgent* and prohibit any time advancement.

The behavior of a TAIO is defined using a current state which is a pair of: a location ℓ, and a set of clock values. The initial state of the example is $\langle \ell_0, \{0, 0\} \rangle$, *i.e.* the state at location ℓ_0 with a value of 0 for x and 0 for y. From the initial state, there are two possible kinds of model movements: the input i_2 can be received, then the next state should be $\langle \overline{\ell_1}, \{0, 0\} \rangle$ or time can elapse, say for 0.1 mtu (it cannot elapse more than 1.5 mtu), then the next state should be $\langle \ell_0, \{0.1, 0.1\} \rangle$. Notice that initially the input i_1 cannot be received by the transition ℓ_0 to ℓ_1 because the guard $x \geq 1$ does not hold.

The model Figure 2.7 extends with time the specifications of the location ℓ_0 in the previous example Figure 2.6. Here the time is explicitly detailed. ◊

Let $\mathcal{X}$ be the set of clocks and $v : \mathcal{X} \to \mathbb{R}_{\geq 0}$ be a clock valuation function

over $\mathcal{X}$. We denote $(v+\delta)(x) = v(x) + \delta$ where $x \in \mathcal{X}$ and $\delta \in \mathbb{R}_{\geq 0}$, the clock valuation adding δ, and $[\mathcal{Y} \leftarrow 0]v$ the valuation assigning 0 to $x \in \mathcal{Y}$ and $v(z)$ to all $z \in \mathcal{X}$ for $z \notin \mathcal{Y}$.

Definition 2.2. *A TAIO is a 6-tuple $\langle \mathcal{L}, \ell_0, \mathsf{A}, \mathcal{X}, \mathcal{I}, \mathcal{E} \rangle$ where:*

- *$\mathcal{L}$ is the set of locations with $\mathcal{L}_u \subseteq \mathcal{L}$ the subset of* urgent *locations,*
- *$\ell_0 \in \mathcal{L}$ is the initial location,*
- *A is a set of actions such that $\mathsf{A} = \mathsf{A}_{\mathsf{in}} \mathbin{\mathring{\cup}} \mathsf{A}_{\mathsf{out}} \mathbin{\mathring{\cup}} \{\tau\}$:*
- *$\mathcal{X}$ is the set of clocks,*
- *$\mathcal{I} : \mathcal{L} \to \mathcal{G}(\mathcal{X})$ is the function assigning invariants to locations and*
- *$\mathcal{E}$ is a set of transitions such that $\mathcal{E} \subseteq \mathcal{L} \times \mathcal{G}(\mathcal{X}) \times \mathsf{A} \times \mathcal{U}(\mathcal{X}) \times \mathcal{L}$*

$\mathcal{G}(\mathcal{X})$ is the set of guards on the clocks of the form $x \bowtie \mathsf{c}$ with $x \in \mathcal{X}$, $\mathsf{c} \in \mathbb{N}$ and $\bowtie \in \{\leq, <, >, \geq\}$. $\mathcal{U}(\mathcal{X})$ is the subset of $\mathcal{X}$ to reset.

We denote a transition $\ell \xrightarrow{g,\mu,\mathcal{Y}} \ell'$ and use $v \models g$ to mean that the valuation v satisfies the guard $g \in \mathcal{G}$. The set $\mathcal{U}(\mathcal{X})$ represented by $\mathcal{Y}$ is the subset of $\mathcal{X}$ updated and assigned to the new valuation $v'(\mathcal{X}) = [\mathcal{Y} \leftarrow 0]v$. For the continuation, we let $\mathbb{R}^{\mathcal{X}}_{\geq 0}$ be the valuation of the set of clocks $\mathcal{X}$.

According to the definitions in [64], the semantics of TAIO models is defined with a Timed Input/Output Transition System (TIOLTS). TIOLTSs extend IOLTSs with two kinds of transitions, discrete and temporal transitions, they manage a set of clock valuations within the model's locations and actions. Such model is used to formally define how the TAIOs can behave and how they can communicate and interleave each other.

Definition 2.3. *A TAIO defines a TIOLTS as a tuple $\langle \mathcal{S}, s_0, \mathsf{A}, \mathrm{T}_d, \mathrm{T}_t \rangle$ where:*

- *$\mathcal{S} := \{\langle \ell, v \rangle \in \mathcal{L} \times \mathbb{R}^{\mathcal{X}}_{\geq 0} \mid v \models \mathcal{I}(\ell)\}$ is an infinite set. $\langle \ell, v \rangle$ is a pair of a location and a finite set of clock valuations such that the valuations satisfy the invariant of this location.*
- *s_0 is the initial state $\langle \ell_0, \vec{0} \rangle$, where $\vec{0}$ assigns all the clocks in $\mathcal{X}$ to 0.*
- *T_d is the set of discrete transitions: $\langle \ell, v \rangle \xrightarrow{\mu} \langle \ell', v' \rangle$ iff $\exists \ell \xrightarrow{g,\mu,\mathcal{Y}} \ell' \in \mathcal{E}$ such that $\ell \xrightarrow{\mu} \ell'$, $v \models g \wedge \mathcal{I}(\ell)$, and $v' \models \mathcal{I}(\ell')$ for $v' = [\mathcal{Y} \leftarrow 0]v$,*
- *T_t is the set of temporal transitions: $\langle \ell, v \rangle \xrightarrow{\delta} \langle \ell, v + \delta \rangle$ for $\delta \in \mathbb{R}_{\geq 0}$, iff $\ell \notin \mathcal{L}_u$ and for all $0 \leq \delta' \leq \delta$, $v + \delta' \models \mathcal{I}(\ell)$.*

Note that a TIOLTS associated to a TAIO has an infinite set of states due to the clock valuations domain and can only move through states with two transitions. A discrete transition, which fires a TAIO transition performing an action μ if the valuation v satisfies the guard g of this transition and if the updated valuation v' satisfies the target location's invariant $\mathcal{I}(\ell')$. A temporal transition, which advances time and adds the flown duration δ to

all the clocks valuations if this result satisfies the location's invariant $\mathcal{I}(\ell)$ for every duration $\delta' < \delta$ and if ℓ is not an urgent location.

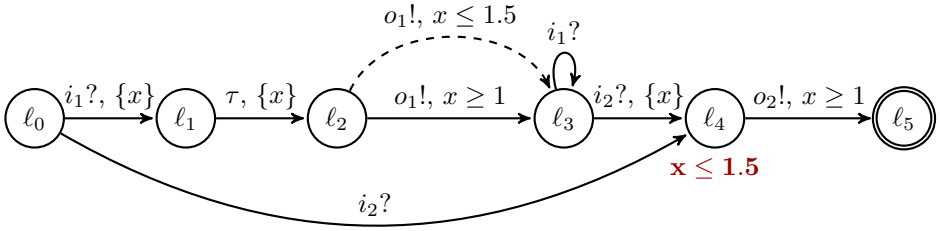

Figure 2.8: TAIO example: playing o_i in a bound of time after receiving the corresponding i_i.

Example 2.2.5: Figure 2.8 depicts a TAIO containing one clock x. This example badly specifies a system that must send an output o_i in the interval of time $[1, 1.5]$ mtu after receiving an input i_i for $i \in \{1, 2\}$. Indeed, assuming our current state in $\langle \ell_0, 1.6 \rangle$, the reception of input i_2 must induce to fire the transition $\ell_0 \xrightarrow{\top, i_2?, \emptyset} \ell_4$. However, this transition cannot be fired because ℓ_4 invariant $(x \leq 1.5)$ does not hold for $x = 1.6$. Hence, the output o_2 cannot be emitted. This situation is called a *deadlock*, at this state no transition is possible anymore. Moreover, x can become greater than 1.5 mtu in location ℓ_2, it is possible because nothing forbids to increment x more than 1.5 mtu. ◊

Definition 2.4. *A finite* path *(or a* run*)* π *of a TAIO is a finite sequence of TIOLTS states obtained from the applications of temporal and discrete transitions, i.e.:*

$$\pi = \langle \ell_0, v_0 \rangle \xrightarrow{\delta_1} \langle \ell_0, v_0 + \delta_1 \rangle \xrightarrow{\mu_1} \langle \ell_1, v_1 \rangle \dots \langle \ell_n, v_n \rangle \xrightarrow{\delta_{n+1}} \langle \ell_n, v_n + \delta_{n+1} \rangle \xrightarrow{\mu_{n+1}} \langle \ell_{n+1}, v_{n+1} \rangle$$

A timed sequence $\nu(\pi)$ *is obtained by projecting the transitions on a* path *(omitting the states), and has the form:* $\nu = \delta_1 \cdot \mu_1 \cdot \ldots \cdot \delta_{n+1} \cdot \mu_{n+1}$. *A* timed trace $\sigma(\pi)$ *is obtained by projecting the transitions on a* path *and accumulating the delays to compute the timestamps* t_i *for each discrete transition. A trace is a sequence of type* $(A_V \times \mathbb{R}_+)*$ *and is computed with* $\langle \alpha_i, \Sigma_{j=1}^{i} \delta_j \rangle$ *for* $1 \leq i \leq n+1$ *and* α_i *a symbol of an observable action on a discrete transition. A trace has the form:* $\sigma = \langle \mu_1, \delta_1 \rangle \cdot \ldots \cdot \langle \mu_{n+1}, \Sigma_{j=1}^{n+1} \delta_j \rangle$.

We define some notations and functions in order to ease the manipulation of a TAIO in the remaining of the document.

Definition 2.5. *Let $\mathcal{T} = \langle \mathcal{S}, s_0, \mathcal{A}, \mathrm{T}_d, \mathrm{T}_t \rangle$ be a TIOLTS. Considering s, $s' \in \mathcal{S}$ and $s_i \in \mathcal{S}$, $\mu_i \in \mathcal{A}$, $\alpha_i \in \mathcal{A}_V$, $o_i \in \mathcal{A}_{\mathsf{out}}$ for all $i \in \{0, 1, \ldots, n\}$, we denote:*

$$
\begin{array}{lcl}
s \rightarrow s' & \stackrel{\text{def}}{=} & \langle \ell, v \rangle \rightarrow \langle \ell', v' \rangle \in \mathrm{T}_d \mathbin{\mathring{\cup}} \mathrm{T}_t \\
s \xrightarrow{\mu} s' & \stackrel{\text{def}}{=} & \langle \ell, v \rangle \xrightarrow{\mu} \langle \ell', v' \rangle \in \mathrm{T}_d \\
s \xrightarrow{\mu} & \stackrel{\text{def}}{=} & \exists\ s'.\ s \xrightarrow{\mu} s' \\
s \not\rightarrow & \stackrel{\text{def}}{=} & \nexists\ s'.\ s \rightarrow s' \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' & \stackrel{\text{def}}{=} & \exists\ s_0, \ldots, s_n.\ s = s_0 \xrightarrow{\mu_0} \ldots \xrightarrow{\mu_n} s_n = s' \\
s \Rightarrow^{\epsilon} s' & \stackrel{\text{def}}{=} & s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \Rightarrow^{\alpha} s' & \stackrel{\text{def}}{=} & \exists\ s_1, s_2.\ s \Rightarrow^{\epsilon} s_1 \xrightarrow{\alpha} s_2 \Rightarrow^{\epsilon} s' \\
s \Rightarrow^{o} s' & \stackrel{\text{def}}{=} & \exists\ s_1, s_2.\ s \Rightarrow^{\epsilon} s_1 \xrightarrow{o} s_2 \Rightarrow^{\epsilon} s' \\
s \Rightarrow^{\nu} s' & \stackrel{\text{def}}{=} & \exists\ s_0, \ldots, s_n.\ s = s_0 \xrightarrow{\delta_1} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\delta_{n+1}} s_{n-1} \xrightarrow{\alpha_{n+1}} s_n = s' \\
 & & \text{with } \nu = \delta_1 \cdot \alpha_1 \cdot \ldots \cdot \delta_{n+1} \cdot \alpha_{n+1}. \\
s \Rightarrow^{\nu} & \stackrel{\text{def}}{=} & \exists\ s'.\ s \Rightarrow^{\nu} s'
\end{array}
$$

- *Let Υ be the set of all the timed traces in $\mathcal{T}$.*
- *$\mathsf{After}(s, \nu) \rightarrow \mathcal{S} \stackrel{\text{def}}{=} \{s' \mid s \Rightarrow^{\nu} s'\}$, for a state s the function After returns all its reachable states after the time sequence ν. We extend this function for timed traces σ accordingly.*
- *$\mathsf{Out}(s) \rightarrow (\mathcal{A}_{\mathsf{out}} \cup \mathbb{R}_+)^* \stackrel{\text{def}}{=} \{o \in \mathcal{A}_{\mathsf{out}} \cup\ \mathbb{R}_+ \mid s \Rightarrow^{o}\}$. The function Out returns the observable outputs and/or durations, that are possible after the state s.*

Definition 2.6. *A TIOLTS is deterministic if: $\forall s \in \mathcal{S}$, if $\exists s_1, s_2.\ \ s \xrightarrow{\mu} s_1$ and $s \xrightarrow{\mu} s_2$ then $s_1 = s_2$.*

In other words a system is deterministic if for all its states every outgoing action has one and only one possible transition. Moreover, to simplify the timed sequences we require:

- time determinism: if $s \xrightarrow{\delta} s'$ and $s \xrightarrow{\delta} s'$ then $s = s'$,

Example 2.2.6: We depict a possible path on the TAIO presented in Figure 2.8:

$$
\begin{aligned}
\pi_1 = & \langle \ell_0, \{0\} \rangle \xrightarrow{0.0} \langle \ell_0, \{0\} \rangle \xrightarrow{i_1?} \langle \ell_1, \{0\} \rangle \xrightarrow{0.4} \langle \ell_1, \{0.4\} \rangle \xrightarrow{\tau} \langle \ell_2, \{0\} \rangle \xrightarrow{0.4} \\
& \langle \ell_2, \{0.4\} \rangle \xrightarrow{0.5} \langle \ell_2, \{0.9\} \rangle \xrightarrow{0.4} \langle \ell_2, \{1.3\} \rangle \xrightarrow{o_1!} \langle \ell_3, \{1.3\} \rangle \xrightarrow{4} \langle \ell_3, \{5.3\} \rangle \xrightarrow{i_2?} \\
& \langle \ell_4, \{0\} \rangle \xrightarrow{1.5} \langle \ell_4, \{1.5\} \rangle \xrightarrow{o_2!} \langle \ell_5, \{1.5\} \rangle.
\end{aligned}
$$

Its related timed sequence and timed trace induce the same sequence of states:

$$\nu(\pi_1) = 0.0 \cdot i_1? \cdot 0.4 \cdot \tau \cdot 0.4 \cdot 0.5 \cdot 0.4 \cdot o_1! \cdot 4 \cdot i_2? \cdot 1.5 \cdot o_2!.$$
$$\sigma(\pi_1) = \langle i_1, 0.0\rangle \cdot \langle o_1, 1.7\rangle \cdot \langle i_2, 5.7\rangle \cdot \langle o_2, 7.2\rangle.$$

Notice that we can formally detect the invariant problem of location ℓ_2. Indeed, $\mathsf{Out}(\langle \ell_2, \{0\}\rangle) = \{o_1!\} \cup \{\mathbb{R}_+\}$ whereas $\mathsf{Out}(\langle \ell_4, \{0\}\rangle) = \{o_1!\} \cup \{1.5\}$, ℓ_2 and ℓ_4 have different behaviors since the time can flow infinitely in ℓ_2 but is bounded to 1.5 in ℓ_4. ◊

2.2.3 Network of Timed Automata.

In order to abstract huge real-time systems, component-based specification [14] decomposes a model into several sub-models. The sub-models are easier to construct than huge models and can specify a concrete system module. Moreover, the different models can share the same set of symbols in order to specify synchronization between them.

We consider a *network* as a composition of several models sharing the same set of actions A. In [17, 14, 10], the *Behavioral, Interaction, Priority* (BIP) operator generalizes the communications in a network model. It compares formal synchronization principles as CCS and SCCS [71, 72] or CSP [57] after defining them with BIP operators. In our case, we define a network of TIOLTSs synchronized with a broadcast CCS method.

Example 2.2.7: Figure 2.9 depicts a network of three TAIO models. The first model emits the symbols i_1 and i_2 following the sequence: $i_1! \cdot 5.7 \cdot i_2!$. i_1 and i_2 are then outputs for this model. The second model is the TAIO example depicted in Figure 2.8. This model waits for i_1 and i_2 which are inputs. The third model similarly waits for i_1 and i_2, and contains a *commit* location to specify an atomic sequence of discrete transitions. When in location C, the next transition of the network should be one of its outgoing transitions, *i.e.* $C \xrightarrow{\top, o_3!, \emptyset} \ell_2$ here. The broadcast communication allows to execute the emission and the two receptions of i_1 simultaneously. One possible sequence in this network is: $i_1 \cdot o_3 \cdot 0.4 \cdot \tau \cdot 0.4 \cdot 0.5 \cdot 0.4 \cdot o_1 \cdot 4 \cdot i_2 \cdot 1.5 \cdot o_2$. Notice that in a network sequence, actions are denoted by their symbols depicting the symbol synchronization between input/output actions. ◊

Let $\mathcal{L}_c \subseteq \mathcal{L}$ be the set of committed locations in a TAIO. A commit location defines the atomicity of a sequence of discrete transitions. Such a location forbids the network to fire any transitions but one of its outgoing

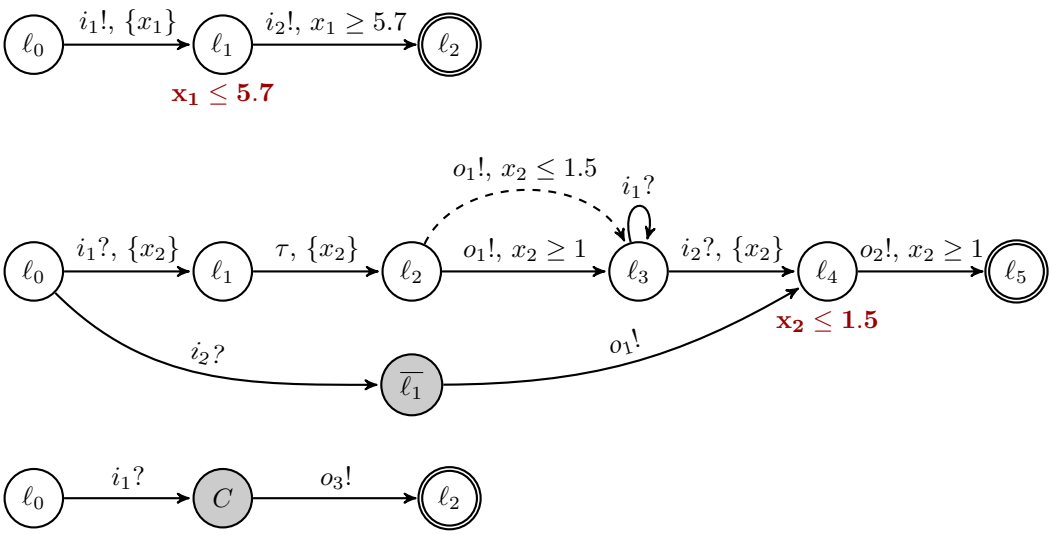

Figure 2.9: An example of a network of three TAIOs. The three models are composed in parallel and communicate with input/output actions.

transitions. Our goal here is to define formally Uppaal networks. Therefore, we present a composition of TIOLTS behaving as an Uppaal network, with broadcast communications and commit locations. The behavior of a network of n TAIOs is defined by a TIOLTS resulting from the parallel composition of n TIOLTS, denoted $\mathsf{T}_1 \| \ldots \| \mathsf{T}_n$. Each TIOLTS in the network is denoted $\mathsf{T}_i\ \langle \mathcal{S}_i, s_{0_i}, \mathsf{A}_i, \mathrm{T}_{d_i}, \mathrm{T}_{t_i} \rangle$ for $i \in 1, \ldots, n$.

Definition 2.7. *The network of n TAIOs defines the TIOLTS $\langle \mathcal{S}, s_0, \mathsf{A}, \mathrm{T}_d, \mathrm{T}_t \rangle$ where:*

- $\mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$,
- $s_0 = s_{0_1} \times \ldots \times s_{0_n}$, *denoted* $\langle s_{0_1}, \ldots, s_{0_n} \rangle$,
- $\mathsf{A} = \cup_{i=1}^{n} \mathsf{A}_i$,
- *for* $i, j, k \in \{1, \ldots, n\}$, T_t *is defined as:*

$$\frac{s_j \xrightarrow{\tau} s'_j}{\langle s_1, \ldots, s_n \rangle \xrightarrow{\tau} \langle \tilde{s_1}, \ldots, \tilde{s_n} \rangle}$$

if $\exists\ s_j = \langle \ell_j, v_j \rangle$ *such that,* $\ell_j \in \mathcal{L}_c$ *or* $\forall s_i = \langle \ell_i, v_i \rangle \in \langle s_1, \ldots, s_n \rangle, \ell_i \notin \mathcal{L}_c$. *Then* $\tilde{s_i} = s'_i$ *if* $i = j$ *and* $\tilde{s_i} = s_i$ *otherwise.*

$$\frac{a = b \qquad i \neq j \qquad s_j \xrightarrow{a!} s'_j \qquad s_i \xrightarrow{b?} s'_i}{\langle s_1, \ldots, s_n \rangle \xrightarrow{a} \langle \tilde{s_1}, \ldots, \tilde{s_n} \rangle}$$

if $\exists\ s_j = \langle \ell_j, v_j \rangle$ *such that,* $\ell_j \in \mathcal{L}_c$ *or* $\forall s_k = \langle \ell_k, v_k \rangle \in \langle s_1, \ldots, s_n \rangle, \ell_k \notin \mathcal{L}_c$*. Then* $\tilde{s_k} = s'_k$ *if* $k = j$ *or* $k = i$ *and* $\tilde{s_k} = s_k$ *otherwise.*

- *for* $i, j, k \in \{1, \ldots, n\}$, T_d *is defined as:*

$$\frac{s_1 \xrightarrow{\delta} s'_1 \qquad \ldots \qquad s_n \xrightarrow{\delta} s'_n}{\langle s_1, \ldots, s_n \rangle \xrightarrow{\delta} (s'_1, \ldots, s'_n \rangle}$$

if $\forall s_i = \langle \ell_i, v_i \rangle \in \langle s_1, \ldots, s_n \rangle, \ell_i \notin \mathcal{L}_c \mathbin{\mathring{\cup}} \mathcal{L}_u$.

A network of n TIOLTSs is n TIOLTSs in parallel on a unified set of actions. Notice that input and output actions may have the same symbols allowing communications between models. A state of the network is a tuple of n TIOLTS states denoted $\langle s_{0_1}, \ldots, s_{0_n} \rangle$. A discrete transition has two cases, an internal or synchronization transition. An internal transition is a discrete transition labeled with τ performed by one TIOLTS. A synchronization of models is one emission by a TIOLTS with none, one or several receptions by others. It implies that all the receptions are done simultaneously when several models wait an emitted symbol. Discrete transitions are fired if none of the TIOLTS states are in a committed location or the current source location (ℓ_i) is committed. A temporal transition advances time for the same duration δ for every TIOLTS. Temporal transitions are fired if none of the TIOLTS states are in a committed or urgent location.

No other restriction is defined, in particular notice that there is an absence of priority between several committed states, and between urgent and normal locations for the synchronization case. These cases are nondeterministic. Moreover a reception transition is fired without regarding the priority of their source locations.

Timed Model Checking. After the construction of a model, one may want to verify if a location is reachable (reachability property), if another is non reachable (safety property), if there exists a path reaching the state before 3 mtu, if after 5 mtu the state is reached for all the possible paths of a model ... These properties are expressible using languages of time logics as in [22] where multiple languages are presented to express such properties. Each language has an expressiveness and a completeness according to their

syntax/semantics and their time interpretation ("pointwise" or "continuous"). See for example the languages TCTL [1] and TPTL [3] which are compared and detailed in [23].

Given a model $\mathcal{M}$ and a correctness property p, model checking consists in deciding whether $\mathcal{M}$ satisfies p. Model checkers are efficient for the verification of real-time systems, and the most famous are Uppaal [44], KRONOS [26] or HyTech [55]. Each framework has proved their applicabilities to industrial systems and contributes greatly to the field of system verification. For the thesis, we decided to use the Uppaal suite tool in order to use an existing and reliable testing framework. According to this choice, we only detail the systems related to these tools.

Uppaal [44] is a framework for designing, simulating and verifying a network of TA. It supplies a useful graphic interface and tools for testing automatically a real time system. Uppaal[1] is based on a network of TA extended with template models, variables and C++ features. The system has been used to verify an amount of industrial applications such as in this non-exhaustive list [24, 41, 69, 42, 61]. In particular, an Audio/Video system [53] and an Audio/Video protocol [25], have been verified using Uppaal and in both of these applications errors have been found and corrected.

For testing, the idea consists in generating a sequence of input traces which covers a set of items of a model $\mathcal{M}$ (some or all its locations, transitions or possible paths). These covering properties of $\mathcal{M}$ can be reduced into reachability problems decidable with model checking.

2.3 Model-Based Testing

Model-Based Testing (MBT) is a *black-box* technique observing exclusively inputs and outputs for testing a system. In [88], MBT is defined as: "a variant of testing that relies on explicit behaviour models that encode the intended behaviour of a system and possibly the behaviour of its environment". In other words, MBT is a general technique for testing a real-time system, called Implementation Under Test (IUT), with respect to a model. This model specifies the good system's behaviors and possibly the test environment that is convenient to delimit test sessions. In our case when dealing with MBT, we always consider the test environment aside system specifications in a model.

Usually and as depicted on the bottom of Figure 2.10, the IUT receives events from a concrete environment (ENV) and reacts by sending actions. A

[1] `http://www.uppaal.org/`

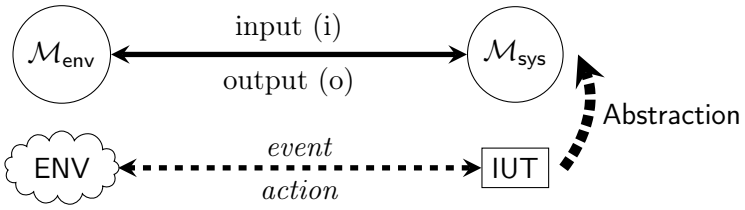

Figure 2.10: Common scheme in model-based testing

first step in MBT consists in the creation of a model $\mathcal{M}$ composed of a IUT specification $\mathcal{M}_{\mathsf{sys}}$ and a model of its test environment $\mathcal{M}_{\mathsf{env}}$. Thereafter, MBT aims at assessing the IUT conformance with respect to the model. Conformance informally means that the concrete input and output exchanges considered in the tests and observed between the IUT and its environment can also be simulated in the model.

Concretely, the conformance is assessed using a huge set of tests called a test suite. A test, created from the model, is defined as two traces: a trace of inputs σ_{in} and its corresponding output trace σ_{ref}. The latter trace is called the reference trace and defines the expected outputs the IUT must send during a simulation following σ_{in}. The concrete output trace returned by the IUT is called a monitored trace σ_{moni}.

A wide number of test frameworks/tools has been designed for testing real-time, probabilist or hybrid systems. As a short list we can cite SpecExplorer[89], TorX[12] and TGV[60]. Usually, a common workflow can be seen and is introduced in Figure 2.11, depicted from system requirements. The system requirements are defined in a document written with a natural language and describing the system's timed behaviors and constraints.

(i) From the system requirements, a model is constructed specifying the system and its test environment.

(ii) The requirements are used to select the test criteria to guide further test generations.

(iii) Then, these criteria are translated into test cases, a formal set of model input and related outputs.

(iv) Thereafter, using the test cases and the model, a suite of tests is concretely generated. A test is a trace of observable inputs σ_{in} (allowed

in the test environment) and expected outputs σ_{ref} computed from a simulation of the specification $\mathcal{M}_{\mathsf{sys}}$.

(v) The IUT is then stimulated according to the input traces of the test suite. This step results in the output trace σ_{moni} observed from the IUT in reaction to each input trace σ_{in}.

(vi) Finally, the test verdict is returned after comparing σ_{moni} and σ_{ref} for each test.

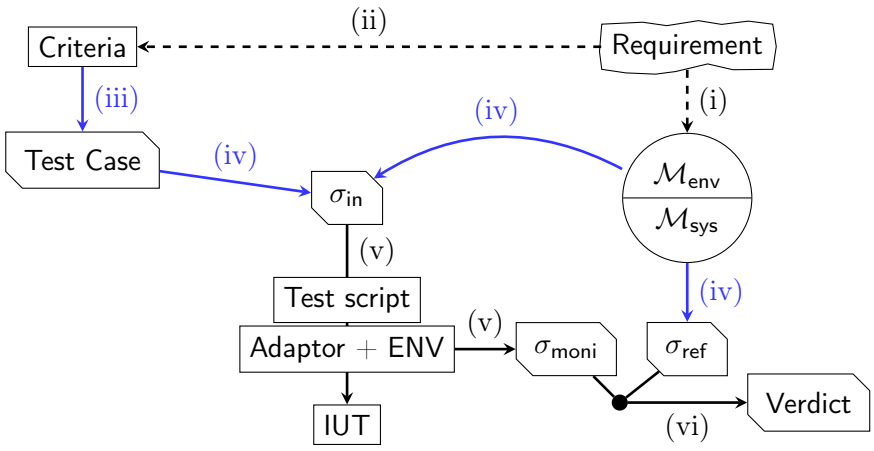

Figure 2.11: The Model-Based Testing Workflow

In short, a MBT application needs a system requirement. The model is usually constructed manually by experts and a common criterion is a transition coverage or a path coverage of the model by the test suite. The test cases are an abstract test suite and defines, for the input sequences of the model $\mathcal{M}$, all its possible outputs and if these outputs are expected. The test cases are then concretized via model simulations with a concrete test suite containing σ_{in} and σ_{ref}. A MBT test terminates on a verdict which pass or fail according to the result of the conformance. If a verdict pass is returned, the IUT conforms the model $\mathcal{M}$, otherwise an unexpected behavior is detected, *i.e.* an IUT error is found.

To formalize system conformance, testing theory defines a set of system's runs as a *test case*. A system's run, called a test, is a suite of input symbols emitted, and output symbols waited. A test starts from the initial state of $\mathcal{M}$ and terminates in a verdict value. A test case is represented with a TIOLTS

[64] (as in Definition 2.3) and, similarly to tests, starts from the initial state of the model $\mathcal{M}$ and terminates in one of the two states: - pass and fail - according to the model expectations. Remark that a test case sends events, hence in $\mathsf{A_{out}}$ and waits for actions in $\mathsf{A_{in}}$.

Definition 2.8. *A test case $\mathcal{T}$ is a TIOLTS $\langle \mathcal{S}, s_0, \mathcal{A}, \mathrm{T}_d, \mathrm{T}_t \rangle$ such that:*

- *$\mathcal{T}$ is deterministic and has a finite behavior,*
- *$\mathcal{S}$ contains terminal states* pass *and* fail*, such that $\ell \not\rightarrow$ for all $\langle \ell, x \rangle \in$ pass $\cup$ fail,*
- *for any state $s \in S$ such that $s \notin$ pass $\cup$ fail, either:*
  - *$s \xrightarrow{e!}$ for some $e \in \mathcal{A}_{\mathsf{out}}$ or*
  - *$\forall a \in \mathcal{A}_{\mathsf{in}}$, $s \xrightarrow{a?}$.*

A test case assures the completeness and the termination of tests by preventing a terminal state from having a possible transition and requiring the test case to be finite. Moreover, a test case requires that in any state either: a wait on all actions of the system is possible, or, one event emission is done. Thus, assuring the test to be non-blocking. Notice that the IUT is assumed non-input-blocking too, *i.e.* the implementation accepts all inputs at any time.

In timed test cases, time is considered as an output, *i.e.* the test case *observes that the system is elapsing some time.* One may abstract time even in un-timed cases, with a dedicated symbol δ, called quiescence, to specify an absence of outputs, interpreted in the test case as: *the system provided no action*, concretely implemented with timeouts.

Example 2.3.8: In Figure 2.12, two test cases are depicted for the TIOLTS model on the top. Notice that the idea is simple and consists in choosing an event to stimulate, and compute following the model, for all the possible outputs, a next state that is either terminal or such that another event can be emitted to continue the test.

Briefly, the test case on the left starts with the emission of input i_1. According to the model, o_1 is expected after 1 time unit. Thus, receiving o_1 when $x < 1$ leads to a terminal state fail. The test case leads to a state fail if o_2 is received too, because the model specifies an emission of o_1 after the input i_1. However, if o_1 is received when $x \geq 1$, the test case continues by sending the second input i_2. Similarly to i_1, the terminal state fail is reached if o_1 is received or an emission of o_2 is received before 1 or after 1.5 mtu. $\Diamond$

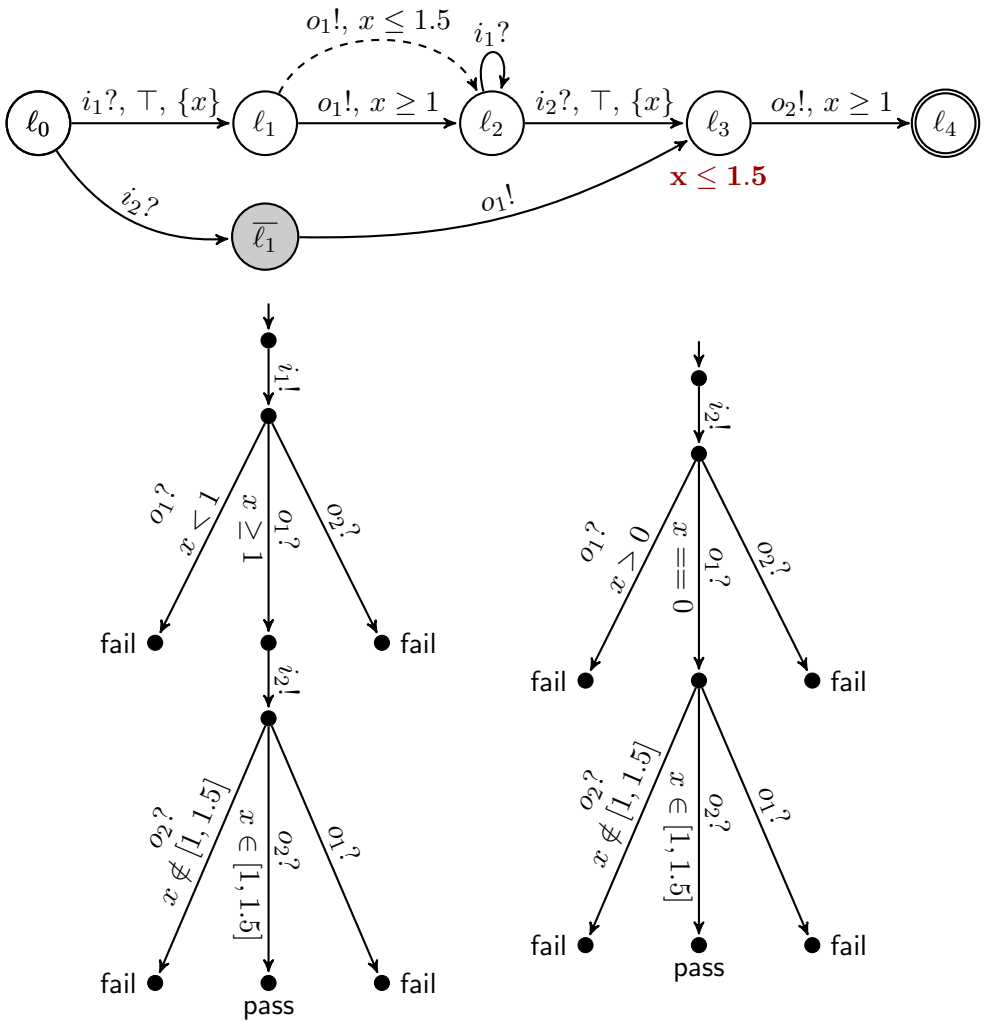

Figure 2.12: The two bottom models are two test cases of the TIOLTS model above. The test cases start at the edge on the top and terminate at one of the terminal states: fail or pass. The internal clock x is reset to 0 by every edge.

Offline and Online Approaches. Given a model $\mathcal{M}$, compute its corresponding suite of input traces $\mathcal{T}_{\mathsf{in}}$ is not an easy task in practice. In particular in case of an exhaustive suite generation, where *all* the possible input traces must be computed. Indeed, the generation of an exhaustive suite induces

to compute the coverage of a trace on a model in order to cover all the model. Two approaches have been designed to tackle the covering problem, an *offline* and *online* approach.

Offline Generation Approach

For MBT, offline approaches first generate the set of input traces $\mathcal{T}_{\mathsf{in}}$. Then, each input trace in $\mathcal{T}_{\mathsf{in}}$ is stimulated to the IUT. The approach has the advantage of returning a covering suite according to a request of model coverage. The input traces generated in $\mathcal{T}_{\mathsf{in}}$ are usually small-sized, that ease the detection of the cause when an error is raised. The traces are fast to execute and adaptable to model variations. However, an offline approach drawback is its complete exploration of models which often implies *state explosions*. A second inconvenient of this approach is the impossibility to deal with non-deterministic models.

In short, offline testing guarantees a suite of input traces with a good coverage but restricts the expressivity of models. Moreover, the approach needs to store the sets of generated traces which can explode in size.

CoVer [18] is an extension of Uppaal for automatically generating a set of input traces covering the model following criteria. Given such criteria (written in a DSL) and a Uppaal model, CoVer translates the criteria into an observer that will guide the offline generation by monitoring the model simulation. Indeed this observer is a finite state machine in which final locations are reached if the corresponding coverable item is effectively reached in the simulation.

Online Generation Approach

The online approach puts in concurrence the on-the-fly simulation of the model and the execution of the IUT. At each step, the algorithm checks IUT outputs, chooses the next input or stops the test, returning a verdict pass in order to validate the test. More precisely, an IUT output is timestamped when received by the model. Then, the outputs are verified according to the conformance relation (*i.e.* if these outputs are currently possible in the model). If the outputs are expected by the model, the test continues, otherwise a verdict fail is returned in order to raise the error. For this algorithm, an input is either: a wait for an amount of time, or an input symbol emission simulated in the model and stimulated to the IUT. Hence, during an online approach, the test passes until an unexpected or an expected but not received output occurs.

Online approach allows nondeterminism in the model and in the strategy of generation. For instance, the generation can randomly choose between a wait, an event emission or a stop/restart. In practice, online tests run longer than offline ones because the testers use the online aspect to run tests during days. In this case, it is more complicated to detect why an error has been raised. For the online generation, no a priory covering algorithm is developed, therefore, the input suite generated has no coverage guarantee. Moreover, due to its real-time context, online testing can hide problems during the process and corrupt the tests (*e.g.* communication delays).

Tron [56, 44] is a second Uppaal extension developed for online testing. Given a model, the observable inputs/outputs and the amount of seconds one mtu lasts, the software (in a client/server manner) runs the online testing process. An interface is given and the IUT must only manage communications via an adaptor which operates the input stimulations and output redirections from and to Tron. To manage real-time problems (due to the network communications or the execution time) a time-server is usable but implies the IUT to be fully time-controllable. In other words, if the IUT manages virtual clock and can simulate a time advancement, the test execution can be synchronized by the time-server which can send amounts of time to elapse to the IUT.

In the following, we present the testing theory for real-time systems. Then, we define the formal conformance testing followed by our MBT framework.

Testing Context. In [87, 85], the Input Output COnformance relation (ioco) is formally defined. The relation ioco allows the conformance testing of two systems, a IUT and its specification, represented with IOLTS models. This conformance is relative to a *class of test*, *observers* and an *implementation relation*. The class of test is defined by the test suite considered in the test, the observers define the manner in monitoring the specification and implementation behaviors of models, and the implementation relation defines the way of comparing two models behaviors. Given two IOLTS, a specification $\mathcal{M}$ and an implementation *iut*, ioco defines: $\mathcal{M}$ is equivalent to *iut* if any test cases in the class of tests lead to the same observations with $\mathcal{M}$ as with *iut*. Formally:

$$iut \ \mathsf{ioco}_{\Upsilon} \ \mathcal{M} \stackrel{\text{def}}{=} \forall_{\nu} \in \Upsilon, \ \mathsf{Out}(\mathsf{After}(iut, \nu)) \subseteq \mathsf{Out}(\mathsf{After}(\mathcal{M}, \nu))$$

for Υ a set of untimed sequences related all the traces of the model and denoting $\mathsf{After}(m, \nu)$ for $\mathsf{After}(q_0, \nu)$ with m a $\mathsf{IOLTS} = \langle \mathcal{Q}, q_0, \mathsf{A}, \mathrm{T} \rangle$.

Timed ioco (tioco) is defined in [64, 80] similarly to its untimed counterpart but interpreting timed traces. In [82], some timed conformance extensions are compared according to multiple ways of managing time flowing. It results that the relation rtioco_e is more expressive (permits to test with more precision), in particular than tioco, since it is related to a test environment, allowing more possibilities for the users. In [56, 44], the definition of timed MBT is based on rtioco_e which is the foundation of Uppaal suite tools for testing real-time systems. Finally, the conformance is defined for two TIOLTSs $\mathcal{M}$ and iut:

$$iut\ \mathsf{rtioco}_e\ \mathcal{M} \iff \forall \nu \in \Upsilon_e,\ \mathsf{Out}(\mathsf{After}(iut, e, \nu)) \subseteq \mathsf{Out}(\mathsf{After}(\mathcal{M}, e, \nu))$$

for Υ_e the set of timed sequences related to the possible traces of the test environment model e and $\mathsf{After}(m, e, \nu)$ the function computing the set of states on the TIOLTS $m \| e$ after the sequence ν with m and e two TIOLTSs.

We consider the same approach in our proper tests and implemented our MBT framework following the conformance rtioco_e.

Conformance Implementation

Our MBT framework, presented in the document, implements the previous conformance definitions and MBT presentations. This section details the gap between formalization and implementation. In particular, we manage timed traces and implemented an extension of the trace conformance as the implementation conformance. Moreover, we denote:

- $\mathsf{Generation} : \mathcal{M}_{\mathsf{env}} \rightarrow \mathcal{T}_{\mathsf{in}}$ to denote the generation of a suite of input traces,
- $\mathcal{M}_{\mathsf{sys}} : \sigma_{\mathsf{in}} \rightarrow \sigma_{\mathsf{ref}}$ to denote the model simulation following the trace σ_{in}, and computing the trace σ_{ref}, and,
- $\mathsf{IUT} : \sigma_{\mathsf{in}} \rightarrow \sigma_{\mathsf{moni}}$ to denote the IUT stimulation following the trace σ_{in}, and monitoring the trace σ_{moni}. This step is the IUT execution.

Finally, let $\mathcal{T}_{\mathsf{in}}$ be the class of test of a test suite, which is in our case the generated suite of input traces σ_{in} possible in $\mathcal{M}_{\mathsf{env}}$. Strictly speaking, the timed MBT conformance is defined:

$$iut\ \mathsf{rtioco}_e\ \mathcal{M} \iff \forall \sigma_{\mathsf{in}} \in \mathcal{T}_{\mathsf{in}},\ \mathsf{IUT}(\sigma_{\mathsf{in}}) = \mathcal{M}_{\mathsf{sys}}(\sigma_{\mathsf{in}}) \tag{2.1}$$

where the suite $\mathcal{T}_{\mathsf{in}}$ is more or less far from Υ_{in}, the set of all the input traces of $\mathcal{M}$.

Finally, a test passes iff *iut* rtioco_e $\mathcal{M}$ for a generated set $\mathcal{T}_{\mathsf{in}}$, otherwise it fails and $\exists\ \sigma_{\mathsf{in}} \in \mathcal{T}_{\mathsf{in}},\ \mathsf{IUT}(\sigma_{\mathsf{in}}) \not\subseteq \mathcal{M}_{\mathsf{sys}}(\sigma_{\mathsf{in}})$. The last definition guarantees that for an exhaustive set $\mathcal{T}_{\mathsf{in}}$, a verdict pass implies that *iut* rtioco_e $\mathcal{M}$ and that errors are absent of the IUT for the input trace set tested.

Following this formalization, the next chapters define a model for IMS (Chapter 3) from which a model-based testing framework is presented (Chapter 4). Then (Chapter 5) details the application of our test framework to a real IMS through several experiments.

Summary

In this chapter, we presented the context and techniques related to our work. In particular, we defined the mixed music context and music systems characteristics. We detailed the two different visions of the authoring and real-time systems in order to position Antescofo, the system tested in our case study. Then, we defined formally relevant models for real-time testing and presented the conformance followed by our MBT framework. Aside, we gave existing tools in the music and testing fields, which can be used on the purpose of testing real-time music systems.

The goal of this work is to provide a formal and efficient method to test real-time music systems. No existing framework has been designed yet in the music field to fulfill this demand and MBT seems to be an efficient procedure for testing such systems. However, although MBT procedure is designed for real-time systems, music systems add more interactions with its time environment than classical systems in the MBT field. In particular, several time scales must be specified within the model. An investigation is done in this work to apply MBT procedures to IMS and if successfully applicable, to provide a framework for testing music systems.

Chapter 3

Event and Time Triggered Model

Modeling a system implies some abstractions to catch aspects of interest without useless complications. In order to specify real-time music systems, we presented in Section 2.2 several models dedicated to certain abstractions.

However, real-time music systems present an original characteristic: they react to musicians according to their pace and must be both event- and time-triggered. To be clear, we mean event-triggered when a system must perform computations or send outputs at some input detection, and time-triggered when such computations or output must be done after an amount of time usually bounded (*i.e.* implemented through timers). Due to the musical time, the management of several time units must be explicitly mentioned in a IMS model. Moreover, score-based systems present a particular notion of determinism and often manage an ordered output set (*e.g.* activating/enabling effects before sending notes). These characteristics are not easily specifiable in existing models, this drawback motivated us to design and formally introduce ad hoc IMS modeling.

We describe in this chapter a way to formalize IMSs. In this aim, we present in Section 3.1 our model called Interactive Real-Time Model (IRTM). IRTMs are sets of Finite State Machines (FSM) dynamically activated and running through logical instants. In order to use existing MBT tools, such as the Uppaal tool suite [44, 56, 18], a translation in a network of TIOLTS was required. This translation and a prove of its soundness are presented in Section 3.2. Finally, to get the full benefit of our model, an implementation of the formal standard semantics is presented in Section 3.3. The implemen-

tation has the form of a Virtual Machine (VM) executing as a byte-code a IRTM following the standard semantics described in Section 3.1.

3.1 Interactive Real-Time Model

This section introduces our dedicated model, called Interactive Real-Time Model (IRTM), to meet the complex requirement of musical modeling and specify real-time music systems. We focus this model on the specification of temporal behaviors of such systems. A temporal behavior of real-time music systems is deterministic but follows a non-deterministic environment inducing relative time and event reactions.

In order to permit this specification, we define ordered sets of symbols and explicit all the possible behaviors of a model. The order reflects the IMS's output priorities and allows to explicitly define the symbol with an higher priority when several are detected simultaneously. This determinism allows us to interpret a model simulation as a function returning the system expected outputs for a given sequence of inputs.

Aside of this determinism, we added non-determinism to specify the test environment model. A model of test environment is efficient to delimit test sessions and defines the set of possible inputs considered in the tests. In order to allow the specification of a set of input sequences, we provide two non-determinism model behaviors: an emission of a set of symbols and a wait for a duration δ in a time interval.

- The principles of IRTMs are introduced in Section 3.1.1.
- IRTM syntax and standard semantics are defined in Section 3.1.2.

3.1.1 Principles of Interactive Real-Time Models

Before defining the IRTM syntax and semantics, we introduce its main principles. We recall that a model $\mathcal{M}$ is divided in a system specification $\mathcal{M}_{\mathsf{sys}}$ and a test environment $\mathcal{M}_{\mathsf{env}}$.

Ordered Input and Output Alphabets: IRTMs are defined over three disjoint and ordered alphabets of discrete symbols:

* Evt, the input symbols (called event and denoted e),
* Act, the output symbols (called action and denoted a) and

* *Sig*, the internal signals used for the communications and synchronizations between sub-components in the specification $\mathcal{M}_{\mathsf{sys}}$.

Moreover, we define that an internal signal has priority over events. The set *Evt* contains events, sent by $\mathcal{M}_{\mathsf{env}}$ and expected by $\mathcal{M}_{\mathsf{sys}}$. On the contrary, the set *Act* gathers actions, returned by $\mathcal{M}_{\mathsf{sys}}$ and caught by $\mathcal{M}_{\mathsf{env}}$. Finally, the set *Sig* specifies communication signals between two model components and are not observable in the input/output traces. However, do not confound *Sig* and the action τ in TIOLTS, here, internal signals are used to specify synchronizations and not internal computations of the IUT. The action τ is not specified in IRTMs. We commonly use α to denote an emission of, or a wait for, an observable symbol in $Evt \mathbin{\mathring{\cup}} Act$, and μ in $Evt \mathbin{\mathring{\cup}} Act \mathbin{\mathring{\cup}} Sig$. We denote as $\mu?$, the wait for a symbol or signal μ, and $\mu!$ the emission of a symbol or signal μ.

Notice that we only consider discrete symbols and signals. Indeed, we are interested in specifying a IUT timed behavior for a given input sequence. Hence, we decided to not manage continuous events. Moreover, a real-time music system is usually designed as shown in Figure 3.1. A IMS waits for continuous or discrete events from the external environment (midi players, sounds, gestures ...) and notices discretely a detection of symbolic events.

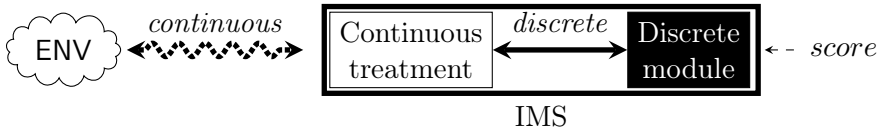

Figure 3.1: IMS (input and output) symbols management

For instance a IMS listening to the audio waves of musicians, uses a score following [35] algorithm in order to map continuous sounds in discrete score events. The discrete symbols are thus already defined and it is not an additional work to consider them in the model.

Logical and Relative Time: Inspired by the synchronous language Esterel [13], a IRTM runs over logical instants. These instants are timestamped with super dense time (from [77, 68]) of the form $\langle t, n\rangle \in \mathbb{R}_{\geq 0} \times \mathbb{N}$, which is a pair of: $t \in \mathbb{R}_{\geq 0}$, a physical timestamp in seconds and $n \in \mathbb{N}$, a number of internal steps. A IRTM behavior can be resumed in:

- advancing time until an event,
- stamping the date to set the beginning of the new instant,

- performing several moves logically instantaneously, incrementing the second field of the super dense timestamp, and
- terminating the instant by advancing time again (when no more moves are possible in the current instant).

Time advancements are specified with delays in IRTMs, each delay relates to a *time unit*. Two kinds of time units are used, the physical and the relative time units.

Definition 3.1. *We let Φ be a relative time unit related to a curve f^{Φ} which defines its pace at a given timestamp. In our context, we call this function a tempo curve, and refer to the pace of a time unit as its tempo in beat per minute (*bpm*). We assume the tempo curve associating an instant tempo value to each timestamp t in physical time. The conversion of a duration d from a relative time unit into physical time is obtained by integration over $[0, d]$ of the inverse of f^{Φ}.*

We denote d^{Φ} a duration d in the relative time unit Φ and phy the physical time unit. The same notation is used for timestamps (or dates) and t^{Φ} is a timestamp in the relative time unit Φ. Moreover, by abuse of notation, we let $\Phi : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ be the evaluation function of Φ. This function evaluates a duration d^{Φ} in a relative time unit into its equivalent duration in physical time d^{phy} for its current *tempo*.

Example 3.1.9: With such durations, the delay 1^{phy} lasts 1 second but 1^{Φ_1} lasts 1 second with a *tempo* of 60_{bpm}, 2 seconds with a *tempo* of 30_{bpm} and 0.5 second with a *tempo* of 120_{bpm}. ◊

Multi time units are quite useful in musical contexts, in the following of the manuscript, we consider relative traces, *i.e.* traces with durations in relative time unit. This definition extends the previous trace Definition 2.4 used in case of traces with durations in physical time.

Definition 3.2. *Given a time unit Φ, a relative timed trace σ^{Φ} is a sequence of triples $\langle \alpha_i, t_i^{\Phi}, p_i^{\Phi} \rangle$, where:*

- *$\alpha_i \in Evt \mathbin{\mathring{\cup}} Act$, is an observable symbol,*
- *$t_i^{\Phi} \in \mathbb{R}_{\geq 0}$, is a relative timestamp, and*
- *$p_i^{\Phi} \in \mathbb{R}_{\geq 0}$, is the tempo value of f^{Φ} at the timestamp t_i^{Φ}.*

Such that for all i, $t_i^{\Phi} \leq t_{i+1}^{\Phi}$ and if $t_i^{\Phi} = t_{i+1}^{\Phi}$ then $p_i^{\Phi} = p_{i+1}^{\Phi}$.

Remark that this definition implies a stepwise constant tempo curve for the relative time units defined by a relative trace. In the following of the document, we use the *performance* time as the time unit which relates to the

tempo curve of musicians (*i.e.* the pace defined by the input trace during a IRTM simulation). It is the time unit used by default and we denote a duration d in performance time with $d^{\sigma_{\text{in}}}$. For simplifying future notations, we denote the evaluation function $\Phi(d)$ to mean d^{phy} if d is in physical time, or $\Phi(d^{\Phi})$ in case of relative durations.

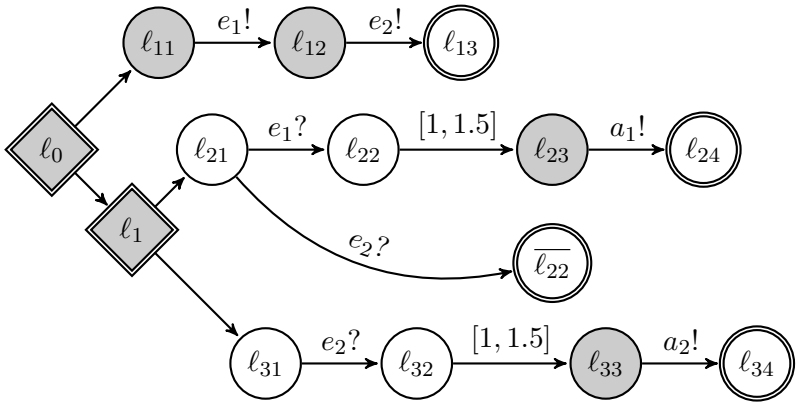

Figure 3.2: IRTM principle: A IRTM is a set of Finite State Machines (FSM) activated dynamically and has one initial location (ℓ_0). The model is mostly deterministic and specifies explicitly time progression through a simulation.

Example 3.1.10: Figure 3.2 depicts an example of IRTM. During simulation, the current model locations are called control points and target a IRTM location together with the amount of time it stayed at this location. The control points are stored in a list and evolve as threads in a cooperative scheduling. That is to say, a control point executes one of the outgoing transitions of its targeted locations while it can.

Initially, the first control point is in the initial location ℓ_0. Diamond locations (ℓ_0 and ℓ_1) depict a transition creating dynamically a control point, after its execution, the first control point is in location ℓ_{11} and a new one is in ℓ_1 and pushed in the list of control points. According to the cooperative scheduling, the first control point keeps the lead. It executes the two next transitions which emit the symbols e_1 and e_2 stored in a set. Then, the control point terminates by deleting itself from the list of control points.

The second control point executes similarly the dynamic creation in ℓ_1. In ℓ_{21}, it can receive two symbols, e_1 and $e2$, both emitted in the current logical instant. It receives the highest priority symbol, say e_1, and is transferred in ℓ_{22}. Because the next transition is a wait for a delay (between 1 and 1.5

performance time), the control point is preempted (called suspended).

The control point in location ℓ_{31} can receive e_2 and is similarly suspended. Every control point is suspended and no more transition can be executed in the logical instant. The instant is terminated and a delay $\delta \in [1, 1.5]$ in performance time is elapsed. At this time, the two control points are unsuspended and can execute the wait transitions in order to send the actions a_1 and a_2. Notice that, because of the list, the control point in location ℓ_{22} has first the lead. $\Diamond$

3.1.2 Syntax and Semantics

This section defines formally the syntax and the main semantics of IRTMs. A IRTM has the form of finite state machines with message passing, dynamic thread creation (alternations) and durations.

IRTM Syntax

Definition 3.3. *A FSM is a tuple* $\langle \Sigma_{in},\ \Sigma_{out},\ \mathcal{L},\ \ell_0,\ \Delta \rangle$ *where:*
- Σ_{in} *is the set of inputs representing the local input alphabet,*
- Σ_{out} *is the set of outputs representing the local output alphabet,*
- $\mathcal{L}$ *is a finite set of locations,*
- $\ell_0 \in \mathcal{L}$ *is the unique initial location and*
- Δ *is a finite set of transitions.*

Each transition in Δ has a source location denoted ℓ, and one or two target locations denoted ℓ' and $\ell_1 \| \ell_2$ respectively. Moreover, $\Delta = \Delta_u \mathbin{\mathring{\cup}} \Delta_s$, the set of transitions is partitioned into: the subset of *urgent* transitions, denoted Δ_u, that must be fired without delay; and the subset of *suspending* transitions, denoted Δ_s, whose execution may require some time to flow.

Symbol priorities. Notice that the alphabets $\Sigma_{in} \cup \Sigma_{out}$ are not necessarily disjoint and we assume a total ordering $\prec$ over $\Sigma_{in} \cup \Sigma_{out}$. This will be used to define a priority for the receptions of symbols in the FSM. We also assume a partition of output symbols into: $\Sigma_{\mathsf{out}} = \Sigma^{\mathsf{sig}}_{\mathsf{out}} \mathbin{\mathring{\cup}} \Sigma^{\mathsf{ext}}_{\mathsf{out}}$. Symbols of $\Sigma^{\mathsf{sig}}_{\mathsf{out}}$ represent internal signals in *Sig*, whereas symbols of $\Sigma^{\mathsf{ext}}_{\mathsf{out}}$ are dedicated to the external environment in *Act*: they are emitted but not captured by the FSM (they are symbols of the output traces σ_{ref}). Recall that *Sig* and *Act* are input and output symbols defined Page 36. Moreover, $\Sigma^{\mathsf{ext}}_{\mathsf{out}} \cap \Sigma_{\mathsf{in}} = \emptyset$. The symbols of $\Sigma^{\mathsf{sig}}_{\mathsf{out}}$ and $\Sigma^{\mathsf{ext}}_{\mathsf{out}}$ will be emitted with different priorities by the FSM, to reflect the semantics of IMS on output actions.

Figure 3.3: *emit*- and *and*-transitions.

Urgent transitions. There are two kinds of *urgent* transitions in Δ_u called *emit*-, and *and*-transitions. They are depicted in Figure 3.3, where circles represent standard locations, diamonds are source-*and* locations and grey locations are instantaneous, so-called *urgent* locations.

1. An *emit*-transition, in $\mathcal{L} \times \Sigma_{out} \times \mathcal{L}$, is denoted $\ell \xrightarrow{\mu!} \ell'$ with $\ell, \ell' \in \mathcal{L}$ and $\mu \in \Sigma_{out}$. It provokes the emission of a local output symbol, followed by the change of the current control point from location ℓ to ℓ'. We say that a location ℓ emits a symbol $\mu \in \Sigma_{out}$ if there exists a transition $\ell \xrightarrow{\mu!} \ell'$ for some location ℓ'. Remark that the source of this transition cannot have another branch, if $\mu \in \Sigma^{\mathsf{ext}}_{\mathsf{out}}$.

2. An *and*-transition, or *alternation*, in $\mathcal{L} \times \mathcal{L}^2$, is denoted $\ell \xrightarrow{\mathsf{and}} \ell_1 \| \ell_2$, with $\ell, \ell_1, \ell_2 \in \mathcal{L}$. It creates dynamically a new control point. The current control point, initially in ℓ, is transferred to the first location ℓ_1, while a new concurrent control point is created in the second location ℓ_2. The source of this transition cannot have another branch.

Figure 3.4: *recv*- and *wait*-transitions.

Suspending transitions. There are also two kinds of *suspending* transitions in Δ_s, called *recv*- and *wait*-transitions. These transitions are depicted in Figure 3.4.

1. A *recv*-transition, in $\mathcal{L} \times \Sigma_{in} \times \mathcal{L}$, is denoted $\ell \xrightarrow{\mu?} \ell'$ with $\ell, \ell' \in \mathcal{L}$ and $\mu \in \Sigma_{in}$. The transition waits for the reception of a local input

symbol, and then changes the current control point from location ℓ to ℓ'. Because μ can already be emitted during the current logical instant, the *recv*-transition may be *urgent* (and may be fired without delay).

2. A *wait*-transition, in $\mathcal{L} \times \mathbb{R}_{>0} \times \mathbb{R}_{>0} \times \mathcal{L}$, is denoted $\ell \xrightarrow{[d,d']} \ell'$, with ℓ and $\ell' \in \mathcal{L}$ and $0 < d \leq d' \in \mathbb{R}_{>0}$ are durations expressed in the same time unit. It waits for the expiration of a delay before changing the control point from location ℓ to ℓ'. Such a transition can only be fired when the control point has spent at least d time units in ℓ. Moreover, it is required that when d' time units have been spent in ℓ, then this transition, or another outgoing one from ℓ must be fired (it is the analogous of TAIO invariants).

Branching transitions. We call a *branch* $\ell \in \mathcal{L}$, all the transitions with the source location ℓ. A branch is the dual of an *and*-transition, and represents the passing of the control point from the source location to one and only one of the target locations of the branch (a branch could be called *or*-transition to this respect).

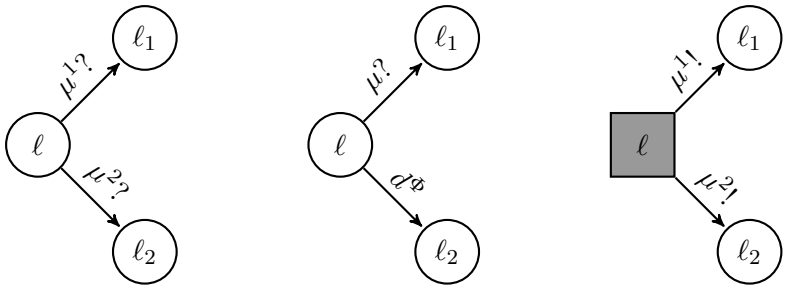

Figure 3.5: Examples of branches.

Example 3.1.11: We depict in Figure 3.5 some examples of branches. Dark rectangle locations depict non-deterministic locations. From left to right we used branch to express: (a) a wait for the first occurrence on several local input symbols, (b) a wait for an event until a timeout, (c) a non-deterministic emission of local output symbols. ♢

A branch will execute *emit*-transitions first, therefore, it makes no sense to branch *suspending* transitions with an *emit*-transition. Moreover, a branch with several *wait*-transitions is relevant if $[d_1, d'_1] \cap [d_2, d'_2] \neq \emptyset$ otherwise the later interval is useless and will be skipped by the earlier. In order to prevent

from *death transitions* which are never executed for any input sequences, we prohibit these two last cases. Notice that the source location of one or several *wait*-transitions has to fire a transition when d' is reached. Hence, only the minimum of all the upper bounds of *wait*-transitions is taken into account, in other words, when there are several *wait*-transitions $[d_i, d'_i]$ in a branch, only $min(d'_i)$ is considered.

Location kinds. The set of locations are partitioned in several disjoint subsets. A location is defined from the types of its outgoing transitions. We define $\mathcal{L} \subseteq \mathcal{L}_u \mathbin{\mathring{\cup}} \mathcal{L}_\infty \mathbin{\mathring{\cup}} \mathcal{L}_s$, with:

(1) $\mathcal{L}_u$, the subset of *urgent* locations, contains source locations with strictly one outgoing *urgent* transition.
(2) $\mathcal{L}_\infty$, the subset of *non-deterministic* locations, contains source locations of several outgoing *emit*-transitions.
(3) $\mathcal{L}_s$, the subset of *suspending* locations, contains the other locations.

IRTM Standard Semantics

Let us now define formally the runs of the above FSMs. We consider a model of superdense time [77, 68] with superdense timestamps of the form $\langle t^{\mathsf{phy}}, n\rangle \in \mathbb{R}_{\geq 0} \times \mathbb{N}$, where $t^{\mathsf{phy}} \in \mathbb{R}_{\geq 0}$ is a timestamp in physical time called logical instant, and $n \in \mathbb{N}$ is a step number inside this logical instant. Intuitively, several transitions may be executed during the same logical instant, at different step numbers. The logical time will flow only when all the control points are in sources of suspending transitions of Δ_s.

Definition 3.4. *A* state *of a FSM* $A = \langle \Sigma_{in},\ \Sigma_{out},\ \mathcal{L},\ \ell_0,\ \Delta\rangle$ *is a tuple* $\langle t^{phy}, n, \Gamma, cp, \Theta\rangle$*, where:*

- $\langle t^{phy}, n\rangle$ *is a super-dense timestamp with* $t^{phy} \in \mathbb{R}_{\geq 0}$ *and* $n \in \mathbb{N}$.
- Γ *is the list of control points of the form* $\langle \ell, \gamma, \beta\rangle$*, where:*
  - $\ell \in \mathcal{L}$*, is a FSM location,*
  - $\gamma : d^{phy} \in \mathbb{R}_{\geq 0}$*, is the time spent in* ℓ*, and,*
  - $\beta \in \{\top, \bot\}$*, is a flag. When* $\beta = \top$*, then the control point is* suspended.
- *cp is a natural number in* $\{1, \ldots, |\Gamma|\}$*, pointing to the current control point in* Γ*, and,*
- Θ *is the subset of local symbols* Σ_{out} *emitted during a logical instant.*

We will later use the operator :: to define both a join operation on the list Γ and an element insertion in the set Θ.

Standard Moves. In the remaining of this section, we will define moves between states of A. The definitions are presented according to the standard semantics, we can distinguish the moves by purposes:

- control point creation and termination: and - exit,
- synchronization: emit - recv - send,
- time management: expir - delay,
- cooperative scheduling: suspend.

To simplify the equations, the current control point (marked by *cp*) is underlined in the following states.

$$\langle t, n, \Gamma :: \underline{\langle \ell, \gamma, \bot \rangle} :: \Gamma', \Theta \rangle \xrightarrow[\mathsf{and}]{} \langle t, n+1, \Gamma :: \underline{\langle \ell_1, 0, \bot \rangle} :: \Gamma' :: \langle \ell_2, 0, \bot \rangle, \Theta \rangle \qquad (\mathsf{and})$$

if there exists $\ell \xrightarrow{\mathsf{and}} \ell_1 \| \ell_2 \in \Delta_u$.

$$\begin{array}{lll} \langle t, n, \Gamma :: \underline{\langle \ell, \gamma, \bot \rangle} :: \langle \ell', \gamma', \beta' \rangle :: \Gamma', \Theta \rangle & \xrightarrow[\mathsf{exit}]{} & \langle t, n+1, \Gamma :: \underline{\langle \ell', \gamma', \beta' \rangle} :: \Gamma', \Theta \rangle \\ \langle t, n, \langle \ell', \gamma', \beta' \rangle :: \Gamma :: \underline{\langle \ell, \gamma, \bot \rangle}, \Theta \rangle & \xrightarrow[\mathsf{exit}]{} & \langle t, n+1, \underline{\langle \ell', \gamma', \beta' \rangle} :: \Gamma, \Theta \rangle \end{array} \qquad (\mathsf{exit})$$

if ℓ is an *exit*-location (has no outgoing transition).

$$\left\langle t, n, \Gamma :: \underline{\langle \ell, \gamma, \bot \rangle} :: \Gamma', \Theta \right\rangle \xrightarrow[\mathsf{emit}]{} \langle t, n+1, \Gamma :: \underline{\langle \ell', 0, \bot \rangle} :: \Gamma', \mu :: \Theta \rangle \qquad (\mathsf{emit})$$

if there exists $\ell \xrightarrow{\mu!} \ell' \in \Delta_u$ with $\mu \in \Sigma^{\mathsf{sig}}_{\mathsf{out}}$.

$$\langle t, n, \Gamma :: \underline{\langle \ell, \gamma, \bot \rangle} :: \Gamma', \Theta \rangle \xrightarrow[\mathsf{expir}]{} \langle t, n+1, \Gamma :: \underline{\langle \ell', 0, \bot \rangle} :: \Gamma', \Theta \rangle \qquad (\mathsf{expir})$$

if (emit) is not applicable and there exists $\ell \xrightarrow{[d,d']} \ell' \in \Delta_s$ such that $\Phi(d) \leq \gamma \leq \Phi(d')$.

$$\langle t, n, \Gamma :: \underline{\langle \ell, \gamma, \beta \rangle} :: \Gamma', \Theta \rangle \xrightarrow[\mathsf{recv}]{} \langle t, n+1, \Gamma :: \underline{\langle \ell', 0, \bot \rangle} :: \Gamma', \Theta \rangle \qquad (\mathsf{recv})$$

if none of (emit) or (expir) can be applied and there exists $\ell \xrightarrow{\mu?} \ell' \in \Delta_s$ such that μ is minimal (wrt $\prec$) in $\Theta \cap \{\mu' \mid \exists \ell \xrightarrow{\mu'?} \ell'' \in \Delta_s\}$.

$$\left\langle t, n, \Gamma :: \underline{\langle \ell, \gamma, \top \rangle} :: \Gamma', \Theta \right\rangle \xrightarrow[\mathsf{send}]{} \langle t, n+1, \Gamma :: \underline{\langle \ell', 0, \bot \rangle} :: \Gamma', a :: \Theta \rangle \qquad (\mathsf{send})$$

if all elements of Γ are suspended and there exists $\ell \xrightarrow{a!} \ell' \in \Delta_u$ with $a \in \Sigma^{\mathsf{ext}}_{\mathsf{out}}$ and a is the smallest symbol of $\Sigma^{\mathsf{ext}}_{\mathsf{out}}$ emitted by a location of Γ.

$$\begin{array}{lll} \langle t, n, \Gamma :: \underline{\langle \ell, \gamma, \beta \rangle} :: \langle \ell', \gamma', \beta' \rangle :: \Gamma', \Theta \rangle & \xrightarrow[\mathsf{susp}]{} & \langle t, n, \Gamma :: \langle \ell, \gamma, \top \rangle :: \underline{\langle \ell', \gamma', \beta' \rangle} :: \Gamma', \Theta \rangle \\ \langle t, n, \langle \ell', \gamma', \beta' \rangle :: \Gamma :: \underline{\langle \ell, \gamma, \beta \rangle}, \Theta \rangle & \xrightarrow[\mathsf{susp}]{} & \langle t, n, \underline{\langle \ell', \gamma', \beta' \rangle} :: \Gamma :: \langle \ell, \gamma, \top \rangle, \Theta \rangle \end{array} \quad (\mathsf{suspend})$$

if none of (and), (exit), (emit), (expir), (recv), (send) can be applied and there exists at least one element in Γ which is not suspended or with a location emitting a symbol of $\Sigma^{\mathsf{ext}}_{\mathsf{out}}$.

$$\langle t, n, \Gamma, \Theta \rangle \xrightarrow[\mathsf{delay}]{} \langle t + \delta, 0, \Gamma + \delta, \emptyset \rangle \quad (\mathsf{delay})$$

if no other move can be applied, where $\Gamma + \delta$ stands for $\{\langle \ell, \gamma + \delta, \bot \rangle \mid \langle \ell, \gamma, \beta \rangle \in \Gamma\}$ and δ a physical time duration such that

a) $\delta > 0$,

b) for all $\langle \ell, \gamma, \beta \rangle$ in Γ such that there exists $\ell \xrightarrow{[d,d']} \ell' \in \Delta_s$, it holds that $\gamma + \delta \leq \Phi(d')$,

c) there exists at least one $\langle \ell, \gamma, \beta \rangle$ in Γ such that $\ell \xrightarrow{[d,d']} \ell' \in \Delta_s$ and $\Phi(d) \leq \gamma + \delta$.

The principle of moves is the following. Every element of Γ represents a thread. Threads run in cooperative scheduling: every thread executes until it gets *suspended* and then it hands over to the next thread in Γ with a move (suspend). During a logical instant, the semantics chooses the moves in the following order:

- Urgent transitions are fired immediately. It gathers the moves (and), (exit) and (emit), except for *emit*-transitions which may be suspended when the symbol to emit is in $\Sigma^{\mathsf{ext}}_{\mathsf{out}}$.
- Except for (expir) moves, which can delay its execution if its upper-bound d' is not reached, suspending transitions must be applied immediately when conditions allow:

(**expir**) A *wait*-transition is applied if the time already spent in the source location, the second component of the current element of Γ, is within the bounds defined for the guard of the transition. Apply first the (expir) moves ensures a priority of time-expirations (*i.e.* delays) over symbol receptions when the both are possible in a branch.

(**recv**) A *recv*-transition is applied if the expected symbol is present in Θ, because it was sent during the same logical instant.

(**suspend**) A failure of these conditions causes the suspension of the thread which changes the value of the flag to $\top$.

(**send**) When all threads are suspended, the *emit*-transitions with symbols of $\Sigma^{\mathsf{ext}}_{\mathsf{out}}$ can be executed, following the ordering $\prec$. This strategy corresponds to the semantics of IMS which requires a predefined ordering for the messages sent to the external environment.

(**delay**) The move (delay) lets a positive amount δ^{phy} of time flow, in adequacy with the upper bounds d' in guards of active *wait*-transitions. A new logical instant is then started at the date timestamped at $t^{\mathsf{phy}} + \delta^{\mathsf{phy}}$, where t^{phy} is the former logical instant, the step counter is reset to zero, the threads of Γ are unsuspended and the list Θ of sent symbols is flushed.

Note that the moves (emit) and (recv) may loop in the same logical instant and that a suspended thread can apply (recv) even if suspended. Moreover, we impose that the delay δ unlocks at least one wait transition, in order to prevent consecutive applications of (delay). The only non-deterministic choices are: the choice of the duration δ in (delay) when $\Phi(d) < \Phi(d')$, and the choice of a local output signal to emit in the set $\Sigma^{\mathsf{sig}}_{\mathsf{out}}$, in (emit), when there are several *emit*-transitions in the same branch. These choices are delegated to the implementation and all the other moves are deterministic.

Deterministic Models. A FSM is called *deterministic* iff it contains no branch with more than one emit transition, and for every wait transition labeled $[d, d']$, it holds that $d = d'$ (we simply write d in this case). Moreover, it must not contain two waits in a branch labeled with the same delay (*i.e.* there must be no transitions $\ell \xrightarrow{d} \ell'$ and $\ell \xrightarrow{d} \ell''$ with $\ell \neq \ell''$).

Run: Starting from the initial state $s_0 = \langle 0, 0, \langle\langle \ell_0, 0, \bot\rangle\rangle, 0, \emptyset\rangle$, a *run* π of a IRTM $\mathcal{A}$ is a sequence of the form:

$$s_0 \xrightarrow[m_1]{} s_1 \dots s_{k-1} \xrightarrow[m_k]{} s_k$$

where $s_0, \dots, s_k$ are states, and for all $0 \leq i < k$, s_{i+1} is obtained from s_i by the move m_{i+1}. We extend TIOLTS's timed sequences ν defined in Section 2.2.1 for IRTM, *i.e.* being only the sequence of moves omitting

intermediate states. Finally, we associate to the run π the output timed trace σ_{ref} defined as follows:

For all $0 \leq i \leq k$, let t_i and Θ_i be respectively the first and the last components of s_i (*i.e.* the timestamp of the logical instant of s_i and the set of accumulated symbols sent during that instant). Let $1 \leq i_1 < \ldots < i_p \leq k$ be the subsequence of all steps in π such that for all $1 \leq j \leq p$, $m_{i_j} = (\mathsf{send})$ and $\Theta_{i_j} \setminus \Theta_{i_j - 1} = \{a_j\} \subset \Sigma^{\mathsf{ext}}_{\mathsf{out}}$, at the move i_j, one output message $a_j \in \Sigma^{\mathsf{ext}}_{\mathsf{out}}$ was emitted. The trace σ_{ref} is a physical time trace and contains the sequence of pairs of the form $\langle a_j, t^{\mathsf{phy}}_{i_j} \rangle$ for all $1 \leq j \leq p$. Given a IRTM $\mathcal{A}$, we denote by $\mathbb{L}(\mathcal{A})$ the set of traces σ_{ref} such that there exists a run π of $\mathcal{A}$ and σ_{ref} is associated to π.

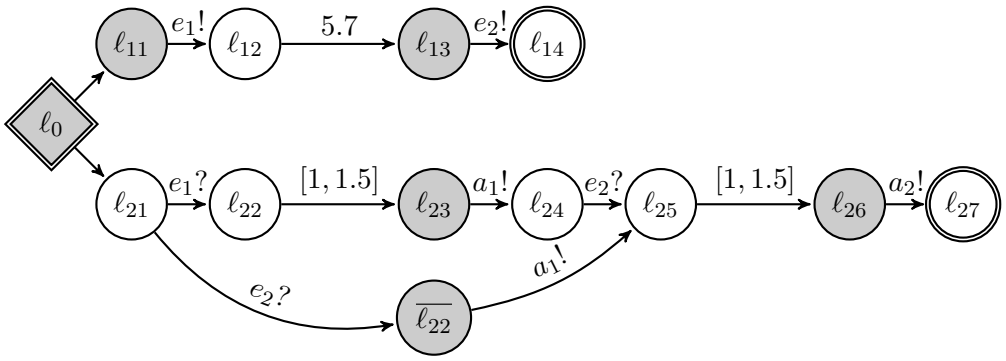

Figure 3.6: IRTM example: This model comes from the TAIO example depicted in Figure 2.8, replacing the input (resp. output) actions by emissions (resp. receptions) of symbols and signals.

Example 3.1.12: The previous example depicted in Figure 2.8 for TAIO models is depicted with a IRTM in Figure 3.6. The bottom FSM has the same behavior as its TAIO counterpart, *i.e.* plays a_i when the corresponding e_i is received after a delay between 1.0 and 1.5 performance time. The FSM on the top of the figure is an environment example, running in parallel (via the *and*-transition from ℓ_0) and only allowing the sequence $\nu = e_1! \cdot 5.7 \cdot e_2!$. Assuming all the delays in the model of type d^{phy}, a simulation will give the path:

$$
\begin{array}{lll}
\langle 0,0,\underline{\ell_{\mathbf{0}}},\emptyset\rangle & \xrightarrow{\text{and}} & \langle 0,1,\underline{\ell_{\mathbf{11}}}::\ell_{21},\emptyset\rangle \xrightarrow{\text{emit}} \\
\langle 0,2,\underline{\ell_{\mathbf{12}}}::\ell_{21},\{e_1\}\rangle & \xrightarrow{\text{suspend}} & \langle 0,2,\breve{\ell}_{12}::\underline{\ell_{21}},\{e_1\}\rangle \xrightarrow{\text{recv}} \\
\langle 0,3,\breve{\ell}_{12}::\underline{\ell_{\mathbf{22}}},\{e_1\}\rangle & \xrightarrow{\text{suspend}} & \langle 0,3,\underline{\breve{\ell}_{\mathbf{12}}}::\breve{\ell}_{22},\{e_1\}\rangle \xrightarrow{\text{delay}^{*1}} \\
\langle 1.3,0,\underline{\ell_{\mathbf{12}}}::\ell_{\mathbf{22}},\emptyset\rangle & \xrightarrow{\text{suspend}} & \langle 1.3,1,\breve{\ell}_{12}::\underline{\ell_{\mathbf{22}}},\emptyset\rangle \xrightarrow{\text{expir}} \\
\langle 1.3,1,\breve{\ell}_{12}::\underline{\ell_{23}},\emptyset\rangle & \xrightarrow{\text{suspend}} & \langle 1.3,1,\underline{\breve{\ell}_{\mathbf{12}}}::\breve{\ell}_{23},\emptyset\rangle \xrightarrow{\text{suspend}} \\
\langle 1.3,1,\breve{\ell}_{12}::\underline{\breve{\ell}_{23}},\emptyset\rangle & \xrightarrow{\text{send}} & \langle 1.3,2,\breve{\ell}_{12}::\underline{\ell_{\mathbf{24}}},\{a_1\}\rangle \xrightarrow{\text{suspend}} \\
\langle 1.3,2,\underline{\breve{\ell}_{\mathbf{12}}}::\breve{\ell}_{24},\{a_1\}\rangle & \xrightarrow{\text{delay}} & \langle 5.7,0,\underline{\ell_{\mathbf{12}}}::\ell_{24},\emptyset\rangle \xrightarrow{\text{expir}} \\
\langle 5.7,1,\underline{\ell_{\mathbf{13}}}::\ell_{24},\emptyset\rangle & \xrightarrow{\text{emit}} & \langle 5.7,2,\underline{\ell_{\mathbf{14}}}::\ell_{24},\{e_2\}\rangle \xrightarrow{\text{exit}} \\
\langle 5.7,3,\underline{\ell_{24}},\{e_2\}\rangle & \xrightarrow{\text{recv}} & \langle 5.7,4,\underline{\ell_{\mathbf{25}}},\{e_2\}\rangle \xrightarrow{\text{suspend}} \\
\langle 5.7,5,\underline{\breve{\ell}_{\mathbf{25}}},\{e_2\}\rangle & \xrightarrow{\text{delay}^{*1}} & \langle 6.7,0,\underline{\ell_{\mathbf{25}}},\emptyset\rangle \xrightarrow{\text{expir}} \\
\langle 6.7,1,\underline{\ell_{26}},\emptyset\rangle & \xrightarrow{\text{suspend}} & \langle 6.7,1,\underline{\breve{\ell}_{26}},\emptyset\rangle \xrightarrow{\text{send}} \\
\langle 6.7,2,\underline{\ell_{\mathbf{27}}},\{a_2\}\rangle & \xrightarrow{\text{exit}} & \langle 6.7,3,[],\{a_2\}\rangle
\end{array}
$$

where a control point $\langle \gamma, \ell, \beta\rangle$ in Γ is depicted by its location ℓ and denoted $\breve{\ell}$ when suspended. Moreover, a control point is painted in:

red when it can apply one of the urgent moves (and), (exit) and (emit),

green if the time spent in its location is sufficient to move via an (expir),

blue if the elements of Θ allow to run with the move (recv),

pink if the next *emit*-transition sends an action ($a \in \Sigma^{\text{ext}}_{\text{out}}$), *i.e.* can apply the move (send) if suspended,

black otherwise.

Finally, ^{*1} highlights the non-determinism due to the interval $[1, 1.5]$ in *wait*-transitions. The resolution (not detailed here) is done during the (delay) moves, and returned first 1.3 seconds and then 1 second in the simulation. Notice that a delay between them is performed to expire the deterministic *wait*-transition with $d = 5.7$ and advances time for 4.4 seconds. The output trace related to this run is: $\sigma_{\text{ref}} = \langle a_1, 1.3\rangle \cdot \langle a_2, 6.7\rangle$. ◊

Example 3.1.13: Assuming now that all the delays in the model are in performance time (of type $d^{\sigma_{\text{in}}}$), the simulation will give a different output. For the relative input trace $\sigma_{\text{in}} = \langle e_1, 0, 120\rangle \cdot \langle e_2, 5.7, 60\rangle$, the delays returned are $0.65s$ (1.3 beat with 120 bpm), $2.2s$ (4.4 beats with 120_{bpm}) and $1s$ (1 beat with 60_{bpm}). It gives the output trace $\sigma_{\text{ref}} = \langle a_1, 0.65\rangle \cdot \langle a_2, 3.85\rangle$, following the input tempi. ◊

To terminate this section, we define a logical instant in a run of IRTM.

Logical instant. A run π exposes the instantaneous sub-sequences of moves executed during a logical instant.

Definition 3.5. *Given a run π of length n, a logical instant π_i of length k is a sub-sequence $s_{i-1} \xrightarrow[m_i]{} s_i \dots s_{i+k-1} \xrightarrow[m_{j+k}]{} s_{i+k}$ with $m_{j-1} = m_{j+k+1} =$ (delay) or $m_j = m_1$ or $m_{j+k} = m_n$.*

A logical instant is a subsequence π_i of a run, it starts the run, terminates the run or is between two (delay) moves. Moreover, the run π is a sequence of l logical instants of the form: $\pi = \pi_1 \xrightarrow[\mathsf{delay}_1]{} \pi_2 \dots \pi_{l-1} \xrightarrow[\mathsf{delay}_{l-1}]{} \pi_l$ where π_i is a logical instant sub-path.

3.2 Correspondence with Timed Automata

Given a IRTM as defined in Section 3.1.2, translations can be performed in order to obtain equivalent existing models. In particular, under some restrictions, a IRTM can be translated into an equivalent network of TIOLTSs (as defined in Section 2.2.1). This translation is concretely performed in our MBT framework to construct a network of TA in the Uppaal format and to use the Uppaal tool suite [44, 56, 18] on these models.

- We define formally a translation from IRTM into a network of TIOLTSs in Section 3.2.1.
- The translation contains several steps using alternative semantics for IRTMs. The soundness of the procedure is proved in Section 3.2.2.

3.2.1 Translation into Timed Automata

We shall first present the restrictions and the logical trace equivalence $\cong$ we consider in the translation. Then, we introduce two alternative semantics for IRTMs, closer to the TIOLTS semantics. Finally, we consider the translation rules and show the soundness of the whole procedure.

Hypotheses and equivalence. The conversion of IRTM into a network of TIOLTSs works under restrictions for the IRTM. These restrictions are required in order to limit IRTMs to what TIOLTSs can express.

(R_1) Only one time unit is supported for the specification of delays.

(R_2) There is no *and*-transition involved in a loop.

The logical trace equivalence $\cong$ is defined for physical output traces of the form $\langle \alpha, t^{\mathsf{phy}} \rangle$. Two physical output traces σ_{out} and σ'_{out} are logically equivalent, written $\sigma_{\mathsf{out}} \cong \sigma'_{\mathsf{out}}$, if for each logical instant the output traces contain the same sets of actions. Formally:

$$\sigma_{\mathsf{out}} \cong \sigma'_{\mathsf{out}} \iff \forall t_i.\ \Sigma^{t_i}_{\mathsf{out}}(\sigma_{\mathsf{out}}) = \Sigma^{t_i}_{\mathsf{out}}(\sigma'_{\mathsf{out}}) \tag{3.1}$$

with $\Sigma^{t_i}_{\mathsf{out}}(\sigma) = \{a_j \mid \langle a_j, t_j \rangle \in \sigma \wedge t_j = t_i\}$. The equivalence is modulo logical instant and does not care of the order of actions with the same timestamp. This relation is extended to the sets of output traces as expected.

(R_3) The output traces $\mathcal{T}_{\mathsf{out}}$ of the IRTM are considered modulo permutations of actions with a same timestamp. The total ordering $\prec$ over Σ_{in} and Σ_{out} is ignored during the comparison of output traces.

The purpose of the restriction R_3 is to fill the gap between the semantics of TIOLTS and the main IRTM semantics. Indeed, there is still a major difference not considered between IRTM and TIOLTS models: the communication mechanisms. IRTM logically and asynchronously sends and receives items using a store Θ (highly inspired from the Esterel signals environment [13]). Contrary to CCS [71, 72] or broadcast in TIOLTS's communications, that are synchronous and without such a Θ-buffer.

Broadcast semantics of IRTM. The idea with the broadcast semantics, is to mimic TIOLTS network behaviors by defining a IRTM semantics using broadcast communication. Let us consider an *alternative semantics* for IRTM, defined like the IRTM standard semantics of Section 3.1.2, except for the following changes:

- The move (emit) is replaced by the following move (broadcast):

$$\langle t, n, \Gamma :: \underline{\langle \ell, \gamma, \top \rangle} :: \Gamma', \Theta \rangle \xrightarrow[\mathsf{broadcast}]{} \langle t, n+k, \tilde{\Gamma} :: \underline{\langle \ell', 0, \bot \rangle} :: \tilde{\Gamma}', \mu :: \Theta \rangle \tag{broadcast}$$

  if all elements of Γ are suspended and there exists $\ell \xrightarrow{\mu!} \ell' \in \Delta_u$ with $\mu \in \Sigma^{\mathsf{sig}}_{\mathsf{out}}$. Moreover, there are $k-1$ running locations with $\langle \ell_i, \gamma_i, \beta_i \rangle$ in $\Gamma \cup \Gamma'$ such that there exists $\ell_i \xrightarrow{\mu?} \ell'_i \in \Delta_s$ and (expir) cannot be executed in ℓ_i. Then, each of them is replaced by $\langle \ell'_i, 0, \bot \rangle$, giving $\tilde{\Gamma} \cup \tilde{\Gamma}'$.

- The move (recv) is replaced by the following move (deadlock):

$$\langle t, n, \Gamma :: \underline{\langle \ell, \gamma, \top \rangle} :: \Gamma', \Theta \rangle \xrightarrow[\mathsf{deadlock}]{} \langle t, n+1, \Gamma :: \underline{\langle \ell', 0, \bot \rangle} :: \Gamma', \Theta \rangle \tag{deadlock}$$

if none of (broadcast) or (expir) can be applied and there exists $\ell \xrightarrow{\mu?} \ell' \in \Delta_s$ with $\mu \in \Theta$.

- The move (send) is replaced by the following move also called (send):

$$\langle t, n, \Gamma :: \underline{\langle \ell, \gamma, \bot \rangle} :: \Gamma', \Theta \rangle \xrightarrow[\mathsf{send}]{} \langle t, n+1, \Gamma :: \underline{\langle \ell', 0, \bot \rangle} :: \Gamma', a :: \Theta \rangle \quad (\mathsf{send})$$

if there exists $\ell \xrightarrow{a!} \ell' \in \Delta_u$ with $a \in \Sigma^{\mathsf{ext}}_{\mathsf{out}}$.

- The move (suspend) is applied if none of (and), (exit) or (send) can be applied.

A (broadcast) move is a synchronous communication by rendez-vous, similar to the communications in the synchronized product of TIOLTS. Roughly, it gathers one (emit) move with none, one or several (recv) moves of Section 3.1.2. However, an important difference is that the control points $\langle \ell, \gamma, \top \rangle$ enabling (emit) and (recv) must all be present in the current vector $\Gamma :: \langle \ell, \gamma, \top \rangle :: \Gamma'$. At the opposite, in the semantics of Section 3.1.2, the (recv) could occur later thanks to the use of the set Θ for storing all the symbols or signals sent during one logical instant.

A (deadlock) move is the reception of a symbol or internal signal μ that cannot be received in an earlier (broadcast) in the logical instant. Moves (deadlock) result in reception transitions not fired in the TIOLTS model because the transition emitting the expected symbol has been already executed, giving a deadlock in TIOLTSs. Intuitively, our goal in this semantics is to avoid the (deadlock) as much as possible. For this purpose, in the alternative semantics, we give to the (broadcast) move a lower priority than its (emit) counterpart in Section 3.1.2 (using the suspend flag). This priority aims at delaying the use of (broadcast) as much as possible and allows all the control points to have the lead before executing a (broadcast) move. It results in firing more *recv*-transitions when the (broadcast) move is executed because other transitions until *recv*-transitions should be executed for all the control points.

Finally, the new (send) move is the same as the (send) of Section 3.1.2, except that it does not care about the order of symbols, and has priority over (broadcast) because it does not require the control point to be suspended.

The move priorities of the alternative semantics are (and), (exit), (send), (suspend), (broadcast), (expir), (deadlock) and (delay). Notice that emissions have still a higher priority than delay expirations, in other words, (expir) moves are applied on suspended control points which take precedence over

receptions, preserving the move applicability order of the standard semantics of Section 3.1.2.

The *broadcast semantics* of IRTM is the same as the alternative semantics, without the (deadlock) move. We consider the same definition of runs as in Section 3.1.2, and given a IRTM $\mathcal{M}$, we denote by $\mathbb{L}_{\mathsf{alt}}(\mathcal{M})$, respectively $\mathbb{L}_{\mathsf{brd}}(\mathcal{M})$, the set of traces σ_{ref} such that there exists a run π of $\mathcal{M}$ following the alternative semantics, respectively broadcast semantics, and σ_{ref} is associated to π.

The alternative semantics is equivalent under the restriction R_3 to the IRTM semantics. Indeed, the (broadcast) move is a sequence of successive (emit) and (recv) moves for the same signal or symbol. In other terms, for all IRTM $\mathcal{M}$, $\mathbb{L}_{\mathsf{alt}}(\mathcal{M}) = \mathbb{L}(\mathcal{M})$. However, this does not hold for the broadcast semantics and there exists some IRTM $\mathcal{M}$ such that $\mathbb{L}_{\mathsf{alt}}(\mathcal{M}) \neq \mathbb{L}_{\mathsf{brd}}(\mathcal{M})$. This inequality is seen for IRTMs executing the move (deadlock) when simulated with the alternative semantics.

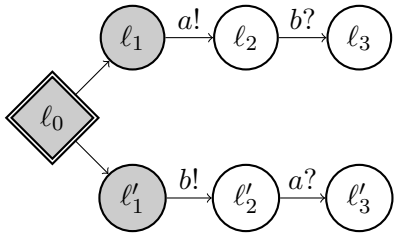

Figure 3.7: A IRTM $\mathcal{M}$ such that $\mathbb{L}_{\mathsf{alt}}(\mathcal{M}) \neq \mathbb{L}_{\mathsf{brd}}(\mathcal{M})$.

Example 3.2.14: Figure 3.7 depicts an example of IRTM $\mathcal{M}$ such that $\mathbb{L}_{\mathsf{alt}}(\mathcal{M}) \neq \mathbb{L}_{\mathsf{brd}}(\mathcal{M})$. Indeed, when the two control points are in locations ℓ_1 and ℓ'_1, the IRTM can send a, with a (broadcast) move. Then, it can both send b (in location ℓ'_1) and receive it (in location ℓ_2), again with a move (broadcast). But thereafter, the IRTM is stuck in ℓ'_2. In order to capture the already sent a, a move (deadlock) would be required, looking in the set Θ of symbols already sent in the same logical instant. We depict the timed sequences for each semantics hiding the (suspend) move executions:

$$\begin{aligned}
\nu &= \mathsf{and} \cdot\ \mathsf{emit}\ a \cdot\ \mathsf{emit}\ b \cdot\ \mathsf{recv}\ a \cdot\ \mathsf{exit} \cdot \mathsf{recv}\ b \cdot\ \mathsf{exit} \\
\nu_{\mathsf{alt}} &= \mathsf{and} \cdot\ \mathsf{broadcast}\ a \cdot\ \mathsf{broadcast}\ b \cdot\ \mathsf{exit} \cdot \mathsf{deadlock}\ a \cdot\ \mathsf{exit} \\
\nu_{\mathsf{brd}} &= \mathsf{and} \cdot\ \mathsf{broadcast}\ a \cdot\ \mathsf{broadcast}\ b \cdot\ \mathsf{exit}
\end{aligned}$$

◊

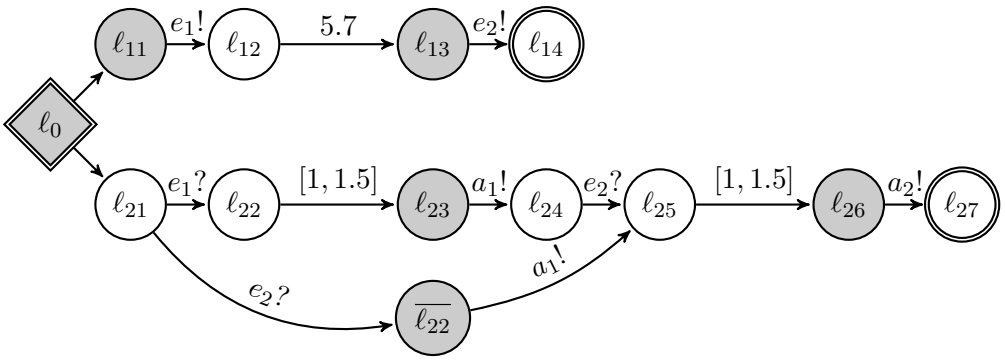

Figure 3.8: IRTM example: This model comes from the TAIO example depicted in Figure 2.9, replacing the input (resp. output) actions by emissions (resp. receptions) of symbols and signals.

Example 3.2.15: Given the IRTM in Figure 3.8 simulated previously, the timed sequence using the alternative semantics is:

$$\nu_{\mathsf{alt}} = \mathsf{and} \cdot \mathsf{broadcast}\ e_1 \cdot\ \mathsf{delay}\ 1.3 \cdot \mathsf{expir} \cdot \mathsf{send}\ a_1 \cdot \mathsf{delay}\ 4.4 \cdot\ \mathsf{expir} \cdot \\ \mathsf{broadcast}\ e_2 \cdot\ \mathsf{exit} \cdot \mathsf{delay}\ 1 \cdot \mathsf{expir} \cdot \mathsf{send}\ a_2 \cdot \mathsf{exit}$$

Because no (deadlock) move is applied for all $\sigma_{\mathsf{ref}} \in \mathcal{T}_{\mathsf{out}}$ (since the possible sequences are $\langle a_1, t_1\rangle \cdot \langle a_2, t_2\rangle$ with $t_1 \in [1, 1.5]$ and $t_2 \in [6.7, 7.2]$) this IRTM is translatable into an equivalent network of TIOLTSs. ◊

Translation of IRTM into TA. Let $\mathcal{M}$ be a IRTM satisfying the restrictions R_1, R_2 and R_3. We show how to construct the corresponding network of TIOLTSs $\mathcal{M}' = \langle \mathcal{S}, s_0, \mathsf{A}, \mathrm{T}_d, \mathrm{T}_t\rangle$ such that $\mathbb{L}_{\mathsf{brd}}(\mathcal{M}) \subseteq \mathbb{L}_{\mathsf{TA}}(\mathcal{M}')$ with $\mathbb{L}_{\mathsf{TA}}(\mathcal{M}')$ the set of traces such that there exists a run π of $\mathcal{M}'$ following the TIOLTS semantics of Section 2.2.1.

The network of TIOLTSs is constructed as the synchronized product of several TIOLTSs, for which their related TAIO $\mathcal{M}'_i = \langle \mathcal{L}_i, \ell_{0_i}, \mathsf{A}_\tau, \mathcal{X}_i, \mathcal{I}_i, \mathcal{E}_i\rangle$ are built from the IRTM. Each $\mathcal{M}'_i$ is over a local single clock x_i and over the alphabet $\mathsf{A}_\tau = \mathit{Evt} \cup \mathit{Act} \cup \mathit{Sig}$, with $\tau \in \mathit{Sig}$ (the input/output symbols and internal signals defined Page 36). The locations of the $\mathcal{M}'_i$ are locations of the IRTM $\mathcal{M}$, plus some fresh locations added below. The transitions and invariants of the $\mathcal{M}'_i$ are built during a traversal of the IRTM $\mathcal{M}$. We

consider that the construction works on a current TAIO (one of the $\mathcal{M}'_i$'s), and present below each step of the traversal:

(1) For an *emit*-transition $\ell \xrightarrow{\mu!} \ell'$ of $\mathcal{M}$, we add to the current TAIO $\mathcal{M}'_i$ a transition $\ell \xrightarrow{\top,\ \mu!,\ \{x_i:=0\}} \ell'$, with, as action, the emission of label μ in the IRTM, without guard and with a reset of the local clock x_i. The source location ℓ is marked as urgent in $\mathcal{M}'_i$ (*i.e.* $\ell \in \mathcal{L}_u$).

(2) We unfold the *and*-transitions of the IRTM into several transitions of concurrent TAIOs. An *and*-transition $\ell \xrightarrow{\mathsf{and}} \ell_1 \| \ell_2$ contains two branches. For the branch $\ell \to \ell_1$, we add to the current $\mathcal{M}'_i$ a transition of the form $\ell \xrightarrow{\top,\ \lambda_{i+1}!,\ \{x_i\ :=\ 0\}} \ell_1$, where λ_{i+1} is a fresh internal signal of A_τ, used to trigger another TAIO and $\ell \in \mathcal{L}_u$. Then we continue the construction of $\mathcal{M}'_i$ starting with the location ℓ_1. When the construction of $\mathcal{M}'_i$ is terminated, we start with a new current TAIO $\mathcal{M}'_{i+1}$ which contains initially a transition $\ell \xrightarrow{\top,\ \lambda_{i+1}?,\ \{x_i\ :=\ 0\}} \ell_2$ (associated to the second branch $\ell \to \ell_2$) and we continue the construction of $\mathcal{M}'_{i+1}$ starting with the location ℓ_2. Remark that the unfolding terminates for all IRTM satisfying restriction R_2.

(3) For a *recv*-transition $\ell \xrightarrow{\mu?} \ell'$ of $\mathcal{M}$, we add to the current TAIO $\mathcal{M}'_i$ a transition $\ell \xrightarrow{\top,\ \mu?,\ \{x_i:=0\}} \ell'$, with, as action, the reception of label μ in the IRTM, without guard and with a reset of the local clock x_i.

(4) For every *wait*-transition $\ell \xrightarrow{[d,d']} \ell'$ of $\mathcal{M}$, we add to the current $\mathcal{M}'_i$ a transition $\ell \xrightarrow{x_i\ \geq\ d,\ \tau,\ \{x_i\ :=\ 0\}} \ell'$ in $\mathcal{E}$, and an invariant $I(\ell) = x_i \leq d'$. In this translation, d' is set as the maximum bound of the local clock x_i in the invariant of its source location $I(\ell)$. This prevents the automaton from staying on the location more than d' mtu. d is set to the minimum bound of x_i into the guards of the transition prohibiting to execute it before d mtu. Notice that when $d = d'$ this combination forces the wait for strictly d mtu.

Example 3.2.16: Figure 3.9 depicts the network of TIOLTSs $\mathcal{M}'$ obtained by translation of the IRTM $\mathcal{M}$ in Figure 3.8. This TIOLTS network is not deterministic and $\mathbb{L}_(\mathcal{M}) \subseteq \mathbb{L}_{\mathsf{TA}}(\mathcal{M}')$. Indeed, the TIOLTS can send a_1 between 1 and 1.5 mtu and a_2 between 5.7 and 7.2 mtu. The IRTM has the same possibilities. ◊

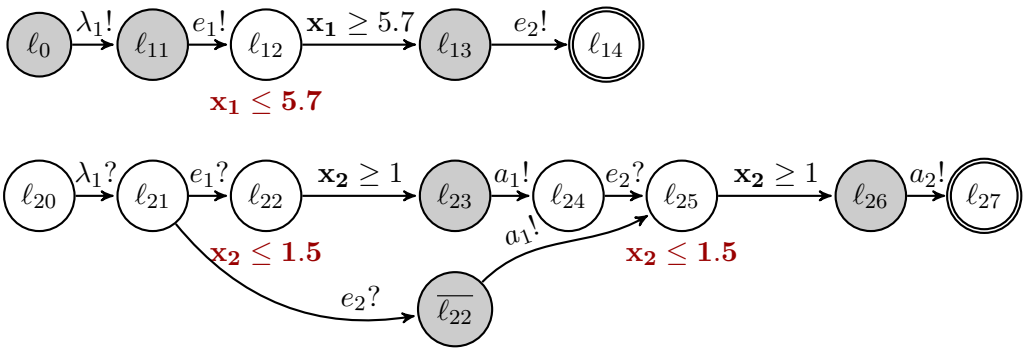

Figure 3.9: TIOLTS network example: Translated from the IRTM example depicted in Figure 2.9. Gray locations are urgent and when omitted, the guard is true ($\top$) and the action is the internal action τ. The constructed TAIOs are *real-time* TAIOs and reset their local clock x_i in every transition. Therefore, for all transitions and for $i \in \{1, 2\}$, $\mathcal{U} = \{x_i\}$.

3.2.2 Soundness of the Translation

Altogether, for a IRTM $\mathcal{M}$ and its TIOLTS translation $\mathcal{M}'$, the wanted property is that $\mathbb{L}(\mathcal{M}) \cong \mathbb{L}_{\mathsf{TA}}(\mathcal{M}')$ holds when $\mathcal{M}$ satisfies R_1, R_2 and R_3. We saw that $\mathbb{L}_{\mathsf{alt}}(\mathcal{M}) \cong \mathbb{L}_{\mathsf{brd}}(\mathcal{M})$ if the model $\mathcal{M}$ does not execute (deadlock) moves. We need to show $\mathbb{L}(\mathcal{M}) \cong \mathbb{L}_{\mathsf{alt}}(\mathcal{M})$ and $\mathbb{L}_{\mathsf{brd}}(\mathcal{M}) \subseteq \mathbb{L}_{\mathsf{TA}}(\mathcal{M}')$.

Proposition 1. $\mathbb{L}(\mathcal{M}) \cong \mathbb{L}_{\mathsf{alt}}(\mathcal{M})$. The idea is based on the same move dependencies in both the alternative and the standard semantics of IRTM. Indeed, for a logical instant t, if a transition is possible for a semantics and fired via a move, the other semantics fires the same transition during the same logical instant t. In particular, for every logical instant $\pi_i = s_{i-1} \xrightarrow[m_i]{} s_i \dots s_{i+k-1} \xrightarrow[m_{j+k}]{} s_{i+k}$ the moves (send) and ($\mathsf{send}_{\mathsf{alt}}$) will output the same set of actions $a_j \in Act$ producing the same set of pairs $\langle a_j, t_i \rangle$ in the related output traces σ_{ref}.

We construct the proposition, in a logical instant and not consider (delay) moves. Moreover, the proposition is presented by recurrence on the possible moves allowed by the standard or alternative semantics, denoted m_i and $m_{i_{\mathsf{alt}}}$ respectively.

The base case holds since we start with the same IRTM $\mathcal{M}$ in state $s_0 = \langle 0, 0, \langle\langle \ell_0, 0, \bot \rangle\rangle, 0, \emptyset \rangle$.

- $\forall \sigma \in \mathbb{L}(\mathcal{M}), \exists \sigma' \in \mathbb{L}_{\mathsf{alt}}(\mathcal{M}), \sigma \cong \sigma'$.

Assuming the model in a state $s_i = \langle t, n, \Gamma :: \langle \ell_{cp}, \gamma_{cp}, \beta_{cp} \rangle :: \Gamma', cp, \Theta \rangle$:

- (and) or (exit) moves are possible. Then, $\ell_{cp} \xrightarrow{\mathsf{and}} \ell_1 \| \ell_2 \in \Delta_u$ or ℓ_{cp} has no outgoing transition and ($\mathsf{and}_{\mathsf{alt}}$) or ($\mathsf{exit}_{\mathsf{alt}}$) can be applied too.
- (emit) is possible, then $\ell_{cp} \xrightarrow{\mu!} \ell'_{cp} \in \Delta_u$ with $\mu \in \Sigma^{\mathsf{sig}}_{\mathsf{out}}$. Moreover:
  1. ($\mathsf{and}_{\mathsf{alt}}$) is not applicable to ℓ_{cp} because an *and*-transition cannot be in a branch,
  2. ($\mathsf{exit}_{\mathsf{alt}}$) cannot be executed because ℓ_{cp} has at least one outgoing transition to ℓ'_{cp},
  3. ($\mathsf{send}_{\mathsf{alt}}$) is not applicable to ℓ_{cp} because an action emission cannot be in a branch.

  Hence, the move ($\mathsf{broadcast}_{\mathsf{alt}}$) can be executed on ℓ_{cp}, when all the control points are suspended. It results in firing the *emit*-transition in the similar fashion than the standard semantics. Remark that we suppose any nondeterminism solved, therefore, another *emit*-transition outgoing from ℓ_{cp} cannot be executed.
- (expir) is possible, then (emit) is not applicable and $\ell_{cp} \xrightarrow{[d,d']} \ell_{cp'} \in \Delta_s$ such that $\Phi(d) \leq \gamma_{cp} \leq \Phi(d')$. From the previous reasons (1), (2) and (3), it implies that ($\mathsf{and}_{\mathsf{alt}}$), ($\mathsf{exit}_{\mathsf{alt}}$), and ($\mathsf{send}_{\mathsf{alt}}$) cannot be executed on ℓ_{cp}. Moreover:
  4. ($\mathsf{broadcast}_{\mathsf{alt}}$) is not possible on ℓ_{cp} because (emit) is not applicable.

  Hence, the move ($\mathsf{expir}_{\mathsf{alt}}$) will be executed on ℓ_{cp}, when all the control points are suspended. It results in firing the *wait*-transition in the similar fashion than the standard semantics. Remark that no ($\mathsf{expir}_{\mathsf{alt}}$) can be skipped by a ($\mathsf{broadcast}_{\mathsf{alt}}$) reception because the reception is prohibited if an expiration is possible from the same source location.
- (recv) is possible, then none of the moves (emit) or (expir) can be applied to ℓ_{cp} and $\ell_{cp} \xrightarrow{\mu?} \ell'_{cp} \in \Delta_s$ such that μ is minimal (wrt $\prec$) in $\Theta \cap \{\mu' \mid \exists \ell_{cp} \xrightarrow{\mu'?} \ell''_{cp} \in \Delta_s\}$. From the previous reasons (1), (2), (3) and (4), it implies that none of ($\mathsf{and}_{\mathsf{alt}}$), ($\mathsf{exit}_{\mathsf{alt}}$), ($\mathsf{send}_{\mathsf{alt}}$) and ($\mathsf{broadcast}_{\mathsf{alt}}$) can be executed on ℓ_{cp}. Moreover:
  5. ($\mathsf{expir}_{\mathsf{alt}}$) is not possible on ℓ_{cp} because (expir) is not applicable.

Hence, the *recv*-transition $\ell_{cp} \xrightarrow{\mu?} \ell'_{cp}$ will be fired on ℓ_{cp} with a future ($\mathsf{broadcast}_{\mathsf{alt}}$) reception if $\mu \notin \Theta$, and with a ($\mathsf{deadlock}_{\mathsf{alt}}$) move otherwise.

- Before ending a logical instant, ($\mathsf{suspend}$) and ($\mathsf{suspend}_{\mathsf{alt}}$) must be executed similarly for all the control points in Γ.
- (send) is possible, then all elements of Γ are suspended and $\ell_{cp} \xrightarrow{a!} \ell'_{cp} \in \Delta_u$ with $a \in \Sigma^{\mathsf{ext}}_{\mathsf{out}}$ and a is the smallest symbol of $\Sigma^{\mathsf{ext}}_{\mathsf{out}}$ emitted by a location of Γ. From the previous reasons (1) and (2), it implies that ($\mathsf{and}_{\mathsf{alt}}$) and ($\mathsf{exit}_{\mathsf{alt}}$) are not applicable to ℓ_{cp}. Hence, ($\mathsf{send}_{\mathsf{alt}}$) is executed and the *emit*-transition is fired similarly than in the standard semantics.

Finally, no move can be executed in the standard semantics except (delay). The equivalent moves are executed in the alternative semantics, in particular, all the ($\mathsf{send}_{\mathsf{alt}}$) moves related to the ($\mathsf{send}$). These *emit*-transition executions result in two sets for all timestamps t: $\Sigma^t_{\mathsf{out}}(\sigma)$ and $\Sigma^t_{\mathsf{out}}(\sigma')$, such that $\Sigma^t_{\mathsf{out}}(\sigma) = \Sigma^t_{\mathsf{out}}(\sigma')$ for $\sigma \in \mathbb{L}(\mathcal{M})$ and $\sigma' \in \mathbb{L}_{\mathsf{alt}}(\mathcal{M})$.

- $\forall \sigma \in \mathbb{L}_{\mathsf{alt}}(\mathcal{M}), \exists \sigma' \in \mathbb{L}(\mathcal{M}), \sigma \cong \sigma'$.
  Assuming the model in a state $s_i = \langle t, n, \Gamma :: \langle \ell_{cp}, \gamma_{cp}, \beta_{cp} \rangle :: \Gamma', cp, \Theta \rangle$:
  - ($\mathsf{and}_{\mathsf{alt}}$) or ($\mathsf{exit}_{\mathsf{alt}}$) moves are possible. Then, $\ell_{cp} \xrightarrow{\mathsf{and}} \ell_1 \| \ell_2 \in \Delta_u$ or ℓ_{cp} has no outgoing transition and (and) or (exit) can be applied too.
  - ($\mathsf{send}_{\mathsf{alt}}$) is possible, then $\ell_{cp} \xrightarrow{a!} \ell'_{cp} \in \Delta_u$ with $a \in \Sigma^{\mathsf{ext}}_{\mathsf{out}}$. From the previous reason (3), it implies that no other move is possible in ℓ_{cp} except ($\mathsf{suspend}$). Hence, (send) is executed, when all the control points are suspended, and the *emit*-transition is fired similarly than in the alternative semantics.
  - Before ending a logical instant, ($\mathsf{suspend}_{\mathsf{alt}}$) and ($\mathsf{suspend}$) must be executed similarly for all the control points in Γ.
  - ($\mathsf{broadcast}_{\mathsf{alt}}$) is possible, then all elements of Γ are suspended and $\ell_{cp} \xrightarrow{\mu!} \ell'_{cp} \in \Delta_u$ with $\mu \in \Sigma^{\mathsf{sig}}_{\mathsf{out}}$. There are $k-1$ running locations $\langle \ell_j, \gamma_j, \beta_j \rangle$ in $\Gamma \cup \Gamma'$ such that there exists $\ell_j \xrightarrow{\mu?} \ell'_j \in \Delta_s$ and (expir) cannot be executed in ℓ_j. Then, each of them is replaced by $\langle \ell'_j, 0, \bot \rangle$, giving $\tilde{\Gamma} \cup \tilde{\Gamma'}$. From the previous reasons (1) and (2), it implies that (and) and (exit) are not applicable to ℓ_{cp}. Hence,

the move (emit) can be executed on ℓ_{cp} and can fire the *emit*-transition in a similar fashion than the alternative semantics. In particular, the execution adds to Θ the symbol μ, enabling the $k-1$ (recv) moves.

- ($\mathsf{expir}_{\mathsf{alt}}$) is possible, then ($\mathsf{broadcast}_{\mathsf{alt}}$) is not applicable and $\ell_{cp} \xrightarrow{[d,d']} \ell'_{cp} \in \Delta_s$ such that $\Phi(d) \leq \gamma_{cp} \leq \Phi(d')$. From the previous reasons (1) and (2), it implies that (and) and (exit) are not applicable to ℓ_{cp}. Moreover, similarly to reason (4), (emit) is not applicable to ℓ_{cp} because ($\mathsf{broadcast}_{\mathsf{alt}}$) is not possible. Hence, the move (expir) can be executed on ℓ_{cp} and can fire the *wait*-transition.
- ($\mathsf{deadlock}_{\mathsf{alt}}$) is possible, then none of ($\mathsf{broadcast}_{\mathsf{alt}}$) or ($\mathsf{expir}_{\mathsf{alt}}$) can be executed and $\ell_{cp} \xrightarrow{\mu?} \ell'_{cp} \in \Delta_s$ with $\mu \in \Theta$. From the previous reasons (1) and (2), it implies that (and) and (exit) are not applicable to ℓ_{cp}. Moreover, similarly to reasons (3) and (4), (emit) and (expir) are not applicable to ℓ_{cp} because ($\mathsf{broadcast}_{\mathsf{alt}}$) and ($\mathsf{expir}_{\mathsf{alt}}$) are not possible. Hence, (recv) can be executed on ℓ_{cp} and can fire the *recv*-transition.

Finally, no move can be executed in the alternative semantics except ($\mathsf{delay}_{\mathsf{alt}}$). The equivalent moves are executed in the standard semantics, in particular, all the (send) moves related to ($\mathsf{send}_{\mathsf{alt}}$). These *emit*-transition executions result in two sets for all timestamps t: $\Sigma^t_{\mathsf{out}}(\sigma')$ and $\Sigma^t_{\mathsf{out}}(\sigma)$, such that $\Sigma^t_{\mathsf{out}}(\sigma') = \Sigma^t_{\mathsf{out}}(\sigma)$ for $\sigma \in \mathbb{L}(\mathcal{M})$ and $\sigma' \in \mathbb{L}_{\mathsf{alt}}(\mathcal{M})$.

Proposition 2. $\mathbb{L}_{\mathsf{brd}}(\mathcal{M}) \subseteq \mathbb{L}_{\mathsf{TA}}(\mathcal{M}')$. The idea is to show via a simulation that the behaviors of any IRTMs simulated with the broadcast semantic can be simulated by its translated network of TIOLTSs. Given a relation $\mathcal{R}$, a simulation defines the fact that one model can express what another expresses. Formally, $\mathcal{M} \sim \mathcal{M}'$ if $\exists\ \mathcal{R} \subseteq \text{IRTM} \times \mathsf{TIOLTS}$ such that $\langle s_{01}, s_{02}\rangle \in \mathcal{R}$ and $\forall \langle s_1, s_2\rangle \in \mathcal{R}$ and $\nu \in \mathbb{L}_{\mathsf{brd}}(\mathcal{M})$ if $s_1 \xrightarrow{\nu} s'_1$, then $\exists s_2,\ s_2 \xrightarrow{\nu'} s'_2$ such that $\nu' \in \mathbb{L}_{\mathsf{TA}}(\mathcal{M}')$, and $\langle s'_1, s'_2\rangle \in \mathcal{R}$. The relation $\mathcal{R}$ is defined as follows:

- $s_{01} = \langle 0, 0, \langle\langle \ell_0, 0, \bot\rangle\rangle, 0, \emptyset\rangle$ for the IRTM $\mathcal{M} = \langle Evt \cup Sig, Act, \mathcal{L}, \ell_0, \Delta\rangle$ and $s_{02} = s_0$ for the TIOLTS $\mathcal{M}' = \langle \mathcal{S}, s_0, \mathsf{A}, \mathrm{T}_d, \mathrm{T}_t\rangle$ with $\langle s_{01}, s_{02}\rangle \in \mathcal{R}$.
- For the state $\langle s_1, s_2\rangle$, with $s_1 = \langle t, n, \Gamma :: \langle \ell_{cp}, \gamma_{cp}, \beta_{cp}\rangle :: \Gamma', cp, \Theta\rangle$, if:

– $s_1 \xrightarrow[\mathsf{and}]{} s_1'$, then $\ell_{cp} \xrightarrow{\mathsf{and}} \ell_1 \| \ell_2 \in \Delta_u$ and:

$$s_1' = \langle t, n+1, \Gamma :: \underline{\langle \ell_1, 0, \bot \rangle} :: \Gamma' :: \langle \ell_2, 0, \bot \rangle, \Theta \rangle$$

By translation (step 2), it exists a related transition in the TAIOs of $\mathcal{M}'$ such that $\ell_i \xrightarrow{\top,\ \lambda_{i+1}!,\ \{x_i := 0\}} \ell_{i'}$ and the created transition $\ell_j \xrightarrow{\top,\ \lambda_{i+1}?,\ \{x_i := 0\}} \ell_{j'}$. Hence, $\mathcal{M}'$ can execute the transition $s_2 \xrightarrow{\lambda_{i+1}} s_2'$, and $\langle s_1', s_2' \rangle \in \mathcal{R}$.

– $s_1 \xrightarrow[\mathsf{exit}]{} s_1'$, then ℓ_{cp} is an *exit*-location (has no outgoing transition). No transition is added in *exit*-locations during translation so for the corresponding TAIO location ℓ_i of s_2, it holds that $\forall \mu \in \mathsf{A}_\tau,\ \ell_i \not\xrightarrow{\mu}$, moreover, $\langle s_1', s_2 \rangle \in \mathcal{R}$.

– $s_1 \xrightarrow[\mathsf{send}]{} s_1'$, then $\ell_{cp} \xrightarrow{a!} \ell' \in \Delta_s$ with $a \in \Sigma^{\mathsf{ext}}_{\mathsf{out}}$. By translation of *emit*-transitions (step 1) and assuming that the TAIO $\mathcal{M}_{\mathsf{env}} \in \mathcal{M}'$, which models the test environment, is input-enable (*i.e.* accepts all the possible outputs at every location in order to prevent from blocking the specification $\mathcal{M}_{\mathsf{sys}}$). It exists in s_2 a corresponding transition $\ell_i \xrightarrow{\top,\ a!,\ \{x_i := 0\}} \ell_{i'}$ with $a \in \mathsf{A}_{\mathcal{O}}$ such that $\mathcal{M}'$ can execute the transition $s_2 \xrightarrow{a} s_2'$ and $\langle s_1', s_2' \rangle \in \mathcal{R}$.

– $s_1 \xrightarrow[\mathsf{suspend}]{} s_1'$, then none of (and), (exit) and (emit) can be executed and there exists at least one element in Γ which is not suspended. This transition explicits the scheduling of IRTM control points, it is considered as non-deterministic in a network of TIOLTSs. Indeed, any TIOLTS states in s_2 can fire a transition if its guard holds. As a consequence, a network of TIOLTSs is more permissive than its related IRTM, and $\mathbb{L}_{\mathsf{brd}}(\mathcal{M}) \subseteq \mathbb{L}_{\mathsf{TA}}(\mathcal{M}')$. Moreover, $\langle s_1', s_2 \rangle \in \mathcal{R}$.

– $s_1 \xrightarrow[\mathsf{broadcast}]{} s_1'$, then all elements of Γ are suspended and $\ell_{cp} \xrightarrow{\mu!} \ell'_{cp} \in \Delta_u$ with $\mu \in \Sigma^{\mathsf{sig}}_{\mathsf{out}}$, there are $k-1$ running locations $\langle \ell_j, \gamma_j, \beta_j \rangle$ in $\Gamma \cup \Gamma'$ such that there exists $\ell_j \xrightarrow{\mu?} \ell'_j \in \Delta_s$ and (expir) cannot be applied to ℓ_j then each of them is replaced by $\langle \ell'_j, 0, \bot \rangle$, giving $\tilde{\Gamma} \cup \tilde{\Gamma}'$. By translation (step 1 and 3), there are:

  * one related transition $\ell_i \xrightarrow{\top,\ \mu!,\ \{x_i := 0\}} \ell_{i'}$ and
  * $k-1$ related transition(s) $\ell_j \xrightarrow{\top,\ \mu?,\ \{x_i := 0\}} \ell_{j'}$

  in s_2 such that $\mathcal{M}'$ can execute the transition $s_2 \xrightarrow{\mu} s_2'$ and $\langle s_1', s_2' \rangle \in \mathcal{R}$.

– $s_1 \xrightarrow[\mathsf{expir}]{} s'_1$, then (broadcast) is not applicable and $\ell_{cp} \xrightarrow{[d,d']} \ell'_{cp} \in \Delta_s$ such that $\Phi(d^\Phi) \leq \gamma_{cp} \leq \Phi(d')$. By translation (step 4), we have three points:

  * the related automaton TAIO_i is on the state $\langle \ell_i, \{\gamma'_{cp}\}\rangle \in s_2$,
  * the related location has an invariant on x_i: $I(\ell_i) = x_i \leq d'$ and
  * the related transition has the form of: $\ell_i \xrightarrow{x_i \geq d,\ \tau,\ \{x_i := 0\}} \ell_{i'}$.

  such that $v(x_i) = \gamma'_{cp}$. The value of γ'_{cp} is the duration in mtu stayed in ℓ_i by the control point, because the TIOLTS $\mathcal{M}'$ resets its local clock in every transition. Indeed, $\gamma'_{cp} = \Phi(\gamma_{cp})^{-1}$ with the relative time Φ, hence, since $d \leq \gamma'_{cp} \leq d'$, $\mathcal{M}'$ can execute the transition $s_2 \xrightarrow{\tau} s'_2$ and $\langle s'_1, s'_2\rangle \in \mathcal{R}$. Remark that when a reception and an expiration hold in a branch, the TIOLTS can choose both of them contrary to IRTM which forces the expiration. This fact implies $\mathbb{L}_{\mathsf{brd}}(\mathcal{M}) \subseteq \mathbb{L}_{\mathsf{TA}}(\mathcal{M}')$.

– $s_1 \xrightarrow[\mathsf{delay}]{} s'_1$, then no other move can be executed, $\{\langle \ell, \gamma^{\mathsf{phy}} + \delta^{\mathsf{phy}}, \perp\rangle \mid \langle \ell, \gamma^{\mathsf{phy}}, \beta\rangle \in \Gamma\}$, and δ^{phy} is a physical time duration such that

  a) $\delta > 0$,

  b) for all $\langle \ell, \gamma, \beta\rangle$ in Γ such that there exists $\ell \xrightarrow{[d,d']} \ell' \in \Delta_s$, it holds that $\gamma + \delta \leq \Phi(d')$ and

  c) there exists at least one $\langle \ell, \gamma, \beta\rangle$ in Γ such that $\ell \xrightarrow{[d,d']} \ell' \in \Delta_s$ and $\Phi(d) \leq \gamma + \delta$.

  By translation (step 4), all the upper bounds d' are in source-location's invariants and all the lower bounds d are in guards. A network of TIOLTSs advances time only in mtu, rather than a IRTM that manipulates physical time. However, restriction R1 relates the mtu to the single relative time Φ in IRTM such that $\delta^\Phi = \Phi(\delta)^{-1}$. Therefore, **b)** assures that $\Phi(\gamma+\delta) \leq d'$ and for all $0 \leq \delta' \leq \delta^\Phi$, $v + \delta' \models \mathcal{I}(\ell)$, v being the valuation of each clocks x_i equal to $\Phi(\gamma)^{-1}$. Hence, $\mathcal{M}'$ can execute the move $s_2 \xrightarrow{\delta} s'_2$ and $\langle s'_1, s'_2\rangle \in \mathcal{R}$.

The relation $\mathcal{R}$ shows that a TIOLTS can simulate a IRTM. However, the translated TIOLTS is more expressive and so a bi-simulation is impossible, *i.e.* a IRTM cannot simulate a TIOLTS via this translation.

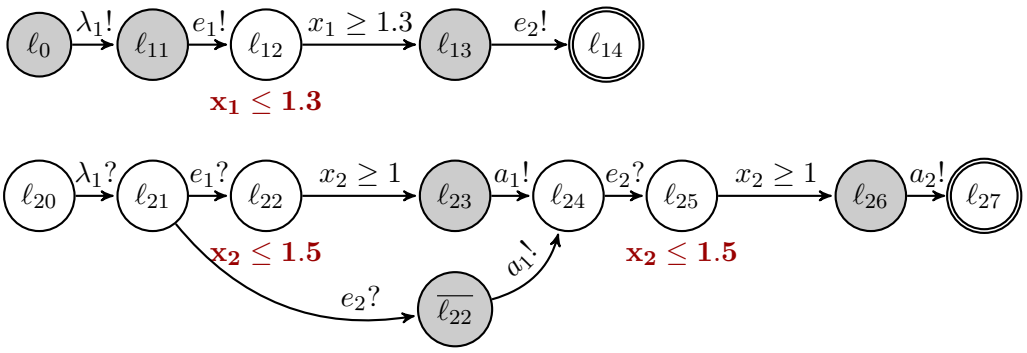

Figure 3.10: Model expressivity: network of TIOLTSs in Figure 3.9 with 1.3 mtu rather than 5.7 mtu.

Example 3.2.17: Figure 3.10 depicts the network of TIOLTSs $\mathcal{M}'$ obtained with a wait for 1.3 mtu in the top *wait*-transition. The example shows that non-bi-simulation is dangerous because $\mathbb{L}_(\mathcal{M}) \subseteq \mathbb{L}_{\mathsf{TA}}(\mathcal{M}')$. Indeed, we can express three different kinds of run:

1) a_1 is sent when $x_2 \in [1, 1.3[$ mtu, then the broadcast with e_2 works and a_2 is sent after $[1, 1.5]$ mtu,

2) a_1 is sent at $x_2 = 1.3$ mtu, then: (a) if the top TAIO executes the emission of e_2 first, the broadcast fails and a_2 is not sent; Otherwise, (b) the same behavior as 1) is seen and a_2 is sent after $[1, 1.5]$ mtu.

3) a_1 is sent when $x_2 \in]1.3, 1.5[$ mtu, e_2 is already sent, therefore, a_2 is not sent because e_2 cannot be received.

A IRTM expresses the first and the second case (a) when a_2 is not sent. However, the second case (b) is impossible with IRTMs because the scheduling executes first the top control point in ℓ_{13}. The third case is possible if the expiration of the top control point does not expire the second. ◊

In conclusion, from a restricted IRTM, the translation constructs a network of TIOLTSs. Due to the nondeterminism we can see in networks of TIOLTSs, the model resulted from translation can express more than its IRTM counterpart. The translation provides a model with more possible paths, implying a generation of more test cases to make an exhaustive suite. However, the exhaustive suite of test cases, covers the test cases generated from the restricted IRTMs.

The restrictions are on the communications, the time units and the alternation managed in IRTMs. More precisely, in order to obtain a communication equivalence between TIOLTSs and IRTMs, (deadlock) moves should not be executed in the set of possible output traces of the IRTM. IRTM delays are restricted to a single time unit to match TIOLTSs model time unit. In particular, a TIOLTS duration d must be a IRTM duration d' after evaluation with $d' = \Phi(d)$. It implies that any shifts of delay performed in a TIOLTS model must consider this tempo curve. Finally, we lost the dynamic concurrence of IRTMs by unfolding *and*-transitions.

However, the TIOLTSs is a good representation of its IRTM counterpart. It allows the integration of efficient and useful frameworks of test in our MBT framework. Moreover, the construction of the TIOLTS model is performed automatically from the IRTM.

3.3 The Real-Time Virtual Machine

IRTMs are defined in order to specify real-time music systems. They can be translated into a network of TIOLTSs to be used by existing MBT tools for generating tests. However, the translation induces several restrictions which may be impossible for some IRTMs. In order to get the full benefit of IRTMs, we present the Virtual Machine (VM) implemented to execute such models. A IRTM execution permits to simulate models without restrictions and thus eases and improves the generation of relevant tests for the IUT. The VM allowed us to implement an online approach in our test framework which can generate a suite of test cases on-the-fly during IRTM simulation. This is a good way to implement efficient algorithms for generating relevant input traces.

Architecture

The VM is composed of four components: its core module, input and output managers and time module. Depicted in Figure 3.11, the VM first requires a IRTM given as input (dashed arrow). Then, the core module executes the model using the other components according to the current executed move or when specific transitions are reached by a control point. We depicted in the figure:

- *step*, to abstract every move of the standard semantics,
- $d?$, to represent activations of *wait*-transitions (*i.e.* when a *wait*-transition is reached by a control point),

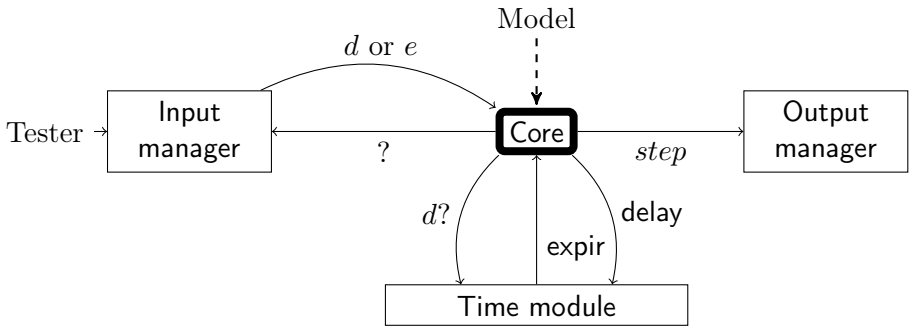

Figure 3.11: VM architecture scheme: given a IRTM as input, the VM simulates the model following the standard semantics.

- ?, to abstract a nondeterministic request,
- d or e, to represent the solution of a nondeterministic request, and finally,
- expir or delay, to depict the corresponding moves.

Core Module Description. The core module of the VM contains the current state $s_i = \langle t_i, n_i,\ \Gamma_i,\ \Theta_i \rangle$ of the simulation. *cp* is implicitly the current control point executing in the list Γ_i. This module applies the possible semantics moves for each control point in Γ_i until their suspension. When a *wait*-transition is reached, delaying the execution by a certain amount of time, a query $d?$ is sent to the time module in charge of waking up the transition when the time has elapsed (resulting in an (expir) move application). When the logical instant is terminated (no more control point can applied a move), a (delay) move informs the time module that time can be advanced. The time module is in charge of computing the shortest δ corresponding to the minimum delay according to the pending *wait*-transitions.

In Γ, the *control points* are managed according to their types: concretely they are stored into the lists Γ_{recv} and Γ_{send}. The former stores the *control points* suspended on a *recv*-transition, and the latter is a priority list sorting the *control points* in a source of a *send*-transition according to the output symbols order. Indeed, a control point is stored only when an event is waited or an action must be sent. The other transitions being *wait*-transitions (stored in the time module) or *urgent* transitions (directly fired), they do not need a specific storage.

Finally, when a nondeterministic location or transition is reached, the core asks the input manager to obtain a value. The solution returned is:

- i) an event e, chosen between the set of possible *emit*-transitions,
- ii) a delay d, picked in the bounds of one or several *wait*-transitions.

The core can then behave deterministically according to the solution, omitting the other possibilities.

Time module. The time module schedules time count downs enabled in the model, *i.e.* the *wait*-duration d retrieved from the time spent on the location. The time count downs are related to the field γ of *control points*. These counters are initialized each time a *control point* has an outgoing *wait*-transition with the value of its delay d (eventually after the resolution of nondeterminism).

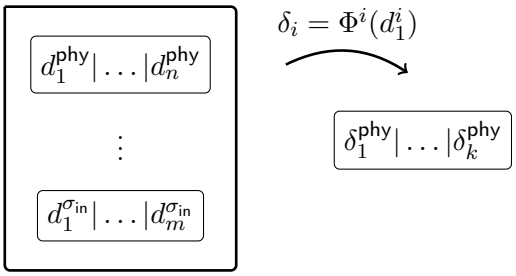

Figure 3.12: Time scheduler scheme.

The counters are sorted according to their types Φ as depicted in Figure 3.12 for k time units. On the left, $k-1$ lists store the enabled count downs for each relative time and one store for physical time. The lists order the earlier duration first and the other relatively. Hence, for the list Φ, a count down d_i^Φ has the value of: $\Sigma_i^{j=1} d_j^\Phi$. Similarly, a delay d^Φ is inserted in the list Φ with the following algorithm:

```
def insert_timer(l, d, i):
  if(d < l.index(i)):
    l.insert(i, d)
    l.index(i+1) = l.index(i+1) - d
  else:
    insert_timer(l, d - l.index(i), i+1)
```

for l a list of count downs of time unit Φ, d the delay to insert of type Φ and i the index.

On the right, a list of delays in physical time is ordered by the k first count downs, allowing to easily compute the shortest delay $\delta = \delta_1^{\mathsf{phy}}$, among all the different time units. In order to sort the different type of delays, we need to apply the evaluation function Φ to the first duration of each list. It permits to manage exclusively physical time. This conversion only needs to be checked every time the move (delay) is executed, since time cannot advance via another move. Notice that the inverse of the evaluation function Φ^{-1} is applied to δ^{phy} in order to retrieve the relative amount of time from the first delay of each list. More precisely, the operation l^{Φ}.index(1) = l^{Φ}.index(1) - $\Phi^{-1}(\delta^{\mathsf{phy}})$ is computed for each list l^{Φ}, waking up the related *wait*-transitions when 0 is reached, (negative values are impossible since δ^{phy} is the shortest delay).

Input-Output managers. The input manager deals with the strategies which emit a sequence of inputs. These emissions depend on the freedom allowed by the model $\mathcal{M}_{\mathsf{env}}$ for performing an input trace. These strategies are executed when a choice is possible and have the responsibility to solve nondeterminism. Different strategies are possible and use "on-the-fly" generation algorithms based on the model $\mathcal{M}$ and possibly some observations. An example of such strategies is implemented and presented in the generation phase in Section 4.3. In short, this strategy returns a random event when several are possible and chooses one of the pending delays δ_i^{phy} (not necessary the first δ_1^{phy}) to return δ when the move (delay) is executed.

The outputs are managed differently since they are independent observers monitoring a precise model aspect. In other words, the output-manager is a class monitoring all the model moves (the *steps* in Figure 3.11) and signaling their application to an observer interface. Thanks to this feature, the observers can choose independently their monitored moves for computing their observations. We implemented three observers:

- the observers tracing the corresponding output trace σ_{ref} or the related input trace σ_{in} of a simulation, catching the send and emit move applications respectively,

- an observer counting the coverage in number of locations of one or several simulations, storing in a set the source locations of every move application,

- and a detailed step printers, mainly used for debugging, which prints the VM internal values.

Summary

This chapter dealt with the formal definition of our ad hoc model to specify event- and time- triggered systems. These models, called Interactive Real-Time Models (IRTM), allow a deterministic timed specification of real-time systems. Their first goal is to be a clear specification of reactive parts of IMSs.

We defined the syntax and the standard semantics of IRTMs allowing such a specification. Then, we presented a translation procedure in order to construct a network of TIOLTSs, behaving as IRTMs. Thereafter, the soundness of the translation is proved, pointing the fact that TIOLTS can express more than IRTMs. The translation is implemented in our framework to use existing MBT frameworks, but is performed under restrictions on IRTMs. We finally presented a VM, implemented in order to consider IRTMs as byte-codes and execute them to return the expected output trace σ_{ref} according to an input trace σ_{in}. The VM allows to generate and simulate without restriction on IRTMs and opens a promising alternative to the translation for testing IMS. First experiments are reported in Section 5.3 with encouraging results.

Chapter 4

Real-Time Model-Based Testing Framework

Now that we have defined a formal model to capture precisely time behaviors of systems, we can present our Timed Model-Based Testing (TMBT) framework. TMBT is based on IRTMs, defined in Chapter 3, and testing theory, presented in Section 2.3. Our framework allows to automatically and formally test real-timed systems given an Implementation Under Test (IUT), its timed requirements and user parameters in inputs.

The goal of this procedure is to check time conformance of real-time systems according to a model $\mathcal{M}$ built from timed requirements. For our framework, an error is a system output not allowed or absent in the model, therefore, we ensure with our tests the system outcomes and *when* they are sent. If a system conforms its model, resulting in a verdict pass, the system's timed reliability should be guaranteed.

This framework avoids the burden of manual construction of models, a disadvantage in usual model-based testing and verification techniques. Indeed, the developed framework proposes an automatic construction based on the high-level timed requirements in input. With such requirements, a MBT procedure is run, checking the Relativized Time Input/Output Conformance (rtioco) [44] between the model and the system outputs. In case of testing score-based IMS, high-level timed requirements are naturally provided by the mixed score needed in such systems and defining the output actions according to the input events of the system.

We present in the following of the section, our TMBT procedure step by step (Section 4.1) in order to describe completely the framework and its possibilities. Then, we describe in Section 4.2 the construction phase, which defines inference rules to make models compositionally. The relevance of the input traces generated for the test is crucial, we investigate this point in Section 4.3 and expose the importance of considering a test environment in models.

4.1 Automatic Model-Based Testing Workflow

We depict in Figure 4.1 TMBT workflow. The workflow requires in input an Abstract Syntax Tree (AST) built from high-level timed *requirements*. In the framework, we consider an abstract syntax used by our *construction* rules for traversing recursively an AST in such a syntax. These rules automatically build the IRTM corresponding to the AST in input. Then, the TMBT workflow, following the MBT approach described in Section 2.3, *generates* a set of input traces $\mathcal{T}_{\mathsf{in}}$ in order to compute for all the input traces $\sigma_{\mathsf{in}} \in \mathcal{T}_{\mathsf{in}}$ the corresponding reference traces σ_{ref} from the model and monitored traces σ_{moni} from the IUT. These traces are computed during the *simulation* and the *execution* steps respectively. Finally, a *comparison* between the reference and the monitored outputs is done resulting in a verdict. This verdict exposes actions per logical instants and reports either the IUT passed or failed the current test suite with the value pass or fail respectively.

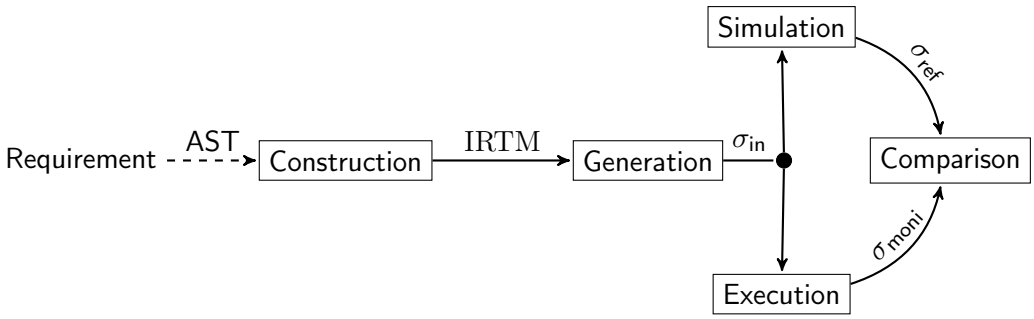

Figure 4.1: Workflow of timed model-based testing framework.

For each step, the procedure permits several configurations and allows the user to control: the test environment $\mathcal{M}_{\mathsf{env}}$, the targeted part of IUT or the test tolerance between the reference and monitored timestamps. The

steps of TMBT workflow are independently configurable, however, there is a global meaning to the reunion of step-configurations. For instance, in order to check the IUT conformance, an exhaustive generation, executed on the whole system must be performed by the framework.

4.1.1 Requirements

The document containing high-level timed *requirements* is the key point for test frameworks and specifies the system features and constraints. In our approach, requirements are considered as a user-friendly document that can be a program, a timed scenario or a list of actions occurring on event detections. In order to generalize these inputs, we directly start from the parsed representation of such a document, a AST following an abstract syntax. The goal is to reduce the cost of model constructions computing a model directly from a list of timed requirements.

requirement	::=	ε | event requirement	
event	::=	evt(e, duration, sequence)	$e \in Evt$
sequence	::=	ε | action sequence	
action	::=	act(duration, sequence) | act(duration, a)	$a \in Act$
duration	::=	d timeUnit	$d \in \mathbb{R}_*^+$
timeUnit	::=	s | t_1	

Figure 4.2: Example of grammar followed by ASTs in input.

Example 4.1.18: For our example, we define the grammar followed by the ASTs depicted in Figure 4.2. This grammar is over two alphabets: Evt and Act. These alphabets define the event and action symbols in the system, *i.e.* the IUT inputs and outputs respectively. We consider in this grammar event- or time-triggered actions, and manage two time units. A duration can be in a relative time defined by the tempo curve Φ_1 and denoted t_1 or in physical time, denoted s.

We denote d, a duration in physical time phy or relative to Φ_1, written d s or d t_1 respectively in the grammar, where d is a duration. The AST given in input must be a sequence of events evt(e, d, as) where $e \in Evt$, d is the event duration and as is the triggered sequence of actions. An action act(d, as) is composed of d a *delay*, and: as a sub-sequence of actions to trigger or, $a \in Act$ an *atomic* action to send. The action delay is the amount of time to wait for until the action activation.

Figure 4.3 depicts a toy example of a textual timed-requirement. Events

are defined with their duration and their related timed reactions (denoted with ->). In this timed requirement, the detection of events e_1 and e_2, must respectively trigger the sequence of actions as_1 and as_2. The first sequence as_1 timely triggers a_1 after 1 second, the second sequence as_2 triggers another actions as_3 0.5 t_1 after the detection of e_2. Then, 0.5 second after this action launch send action a_2. Once again it is an example and the high-level specification is not handled in this work.

$\Diamond$

e_1 1s -> as_1
as_1 -> 1s a_1
e_2 1s -> as_2
as_2 -> 0.5t_1 as_3 0.5s a_2
as_3 -> 1s a_3

Figure 4.3: Example of timed requirement.

Usually, MBT techniques are based on such high-level requirements, used by expert for constructing manually IUT models [88]. Hence, needing such high-level requirements in the input of our framework does not add more work than standard methods and assures a coherence in the test procedures, avoiding misunderstanding and reducing errors in the model. These requirements can then be checked thanks to the graphical representation of models. The intuition here is that writing only once such specification to build the model automatically will ease and reduce errors in testing procedures.

4.1.2 Model Construction

The construction is based on a set of rules parsing the AST. During this step, sub-components of model are created and merged together sequentially or in parallel to build the model $\mathcal{M}$. This modular approach specifies the system reactions for each events/actions and eases the model specification. This approach enjoys the (re)-usability of rules and encourages the scalability.
The construction step, depicted in Figure 4.4, builds a IRTM $\mathcal{M}$ composed of two sub-models: the IUT specification $\mathcal{M}_{\mathsf{sys}}$ and the model of its test environment $\mathcal{M}_{\mathsf{env}}$. The second model sets the bounds of tests that is convenient to delimit test sessions. Indeed, if the test bounds are too large they may lead to an impossible generation of input sequences, but if the test bounds are too restricted they may miss important input sequences not considered

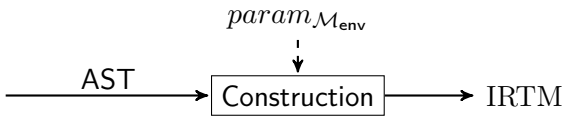

Figure 4.4: TMBT framework - construction step. Given a AST and parameters, a IRTM is constructed specifying the system according to its requirements.

in the tests. The framework requires parameters to construct the wanted kind of environment and allows the user to fix relevant bounds for testing.

More details in the construction rules are presented in Section 4.2. The presentation includes a complete description of the construction rules via toy-examples specifying a system with three different approaches.

4.1.3 Generation of Input Test Data

Given a IRTM $\mathcal{M}$, the generation phase is in charge of producing a suite of input traces $\mathcal{T}_{\mathsf{in}}$. As defined in rtioco_e (Equation 2.1), this set has to be exhaustive in $\mathcal{M}$.

However in practice, exhaustive and even relevant $\mathcal{T}_{\mathsf{in}}$ are difficult or impossible to generate. Indeed, there often is an infinity of possibilities. We implemented several algorithms generating relevant suite of input traces $\mathcal{T}_{\mathsf{in}}$, presented later in Section 4.3. Moreover, another difference with testing theory in Section 2.3, is the concrete stimulation of the IUT by our framework. Testing theory, abstracts the implementation with models to formally define system conformance using test cases as TIOLTS. Here, we manage input and output traces and consider a test case as a pair of traces $\langle \sigma_{\mathsf{in}}, \sigma_{\mathsf{ref}} \rangle$, separating the input and output sequences. From such a pair, every output trace observed from a system stimulated with σ_{in} that diverges from the reference σ_{ref} leads to a fail terminal state.

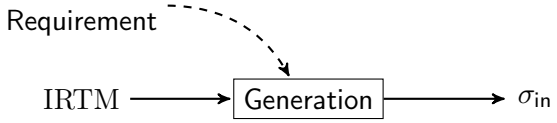

Figure 4.5: TMBT framework - generation step. It generates the set of input traces $\mathcal{T}_{\mathsf{in}}$ according to criteria and $\mathcal{M}_{\mathsf{env}}$ bounds.

As depicted in Figure 4.5, generation may need more information than just the specification. Indeed, high-level requirements or another representation can be more relevant to generate a suite of test. The different generation algorithms are related to the kind of tests the user want to execute, which can cover more or less the model $\mathcal{M}$. The notion of coverage is important to valuate a test suite and its suite of inputs. Generally, a coverage criterion is a reachability problem and must pass by a set of model items: - locations, transitions or paths -, during the simulation of the input trace set $\mathcal{T}_{\mathsf{in}}$. Here, the quality of a test is defined by the coverage of its test case suite.

4.1.4 Simulation for Generation of Reference Output

Giving one or a set of input traces, the simulation step computes the reference traces σ_{ref} gathering the expected outputs of each the related input trace σ_{in}. Simulating a model following an input trace σ_{in} and observing the outputs, results in computing the test case $\langle \sigma_{\mathsf{in}}, \sigma_{\mathsf{ref}} \rangle$.

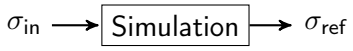

Figure 4.6: TMBT framework - simulation step. The expected outputs traces σ_{ref} are computed by simulating the model with the related input traces σ_{in}.

Figure 4.6 depicts the simulation step. The reference traces σ_{ref} resulted from simulations are physical output traces. The model is required as a finite and non-blocking model for assuring the termination of the simulation.

Our framework simulates two kinds of model:

- Networks of TIOLTSs, in the format of Uppaal, constructed from restricted IRTMs according to the translation presented in Section 3.2. The framework uses the executable CoVer or Verifyta, to run the Uppaal model checker on the model.

- IRTMs, using the VM described in Section 3.3.

In both cases, given σ_{in}, we first generate a deterministic IRTM $\mathcal{M}_{\sigma_{\mathsf{in}}}$ modeling an environment which will strictly follow the input trace. A simulation is then performed by traversing $\mathcal{M}_{\sigma_{\mathsf{in}}}$ or $\mathcal{M}'_{\sigma_{\mathsf{in}}}$ which will send event symbols to the rest of the model $\mathcal{M}_{\mathsf{sys}}$ or $\mathcal{M}'_{\mathsf{sys}}$. Uppaal and the VM, offer options to trace the result in a physical trace σ_{ref} in our format of output traces $\langle a, t^{\mathsf{phy}} \rangle$.

Notice that to translate from Uppaal mtu into physical time, we apply to the durations in σ_{ref} (resulting from Uppaal simulations) the tempo curve Φ from the relative time unit of the restriction R_1 in Section 3.2.

4.1.5 Test Execution

The goal of the execution step is to stimulate and observe the IUT and return the monitored output trace σ_{moni}, which is the system reactions from σ_{in}. Stimulations and observations are done via an *adaptor* which defines *black-box* bounds, *i.e.* the system modules under test. Usually, rather than the entire system, it can be more efficient to restrict the tests only to one or several sub-components of the IUT. For instance, in a IMS testing case, we may want to test only the discrete module of the system.

Indeed, translating the abstract inputs of σ_{in} into concrete ones might be error prone and difficult. Moreover, the reference output trace σ_{ref} might not match the system output anymore. It is due to the impact of the input trace translation, from discrete into concrete inputs, or the system recognition mechanism that blur the input trace σ_{in}. For instance, a score following algorithm [35] usually estimates the tempi of the audio waves during its recognition step. This procedure blurs the tempi in the input trace σ_{in} and alters the system outputs which are not conformed to the related σ_{ref} anymore.

To avoid such problems, the external adaptor which is the interface between the IUT and the model, can directly exchange abstract data with the IUT. It requires two abstract input/output functions to the IUT. In addition to the ease of the tests implementation, these functions allow to compare the system behaviors according to abstract or concrete inputs.

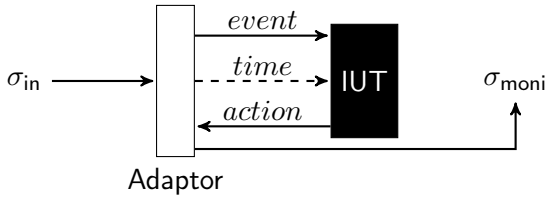

Figure 4.7: TMBT framework - execution step. Adaptors stimulate σ_{in} on the IUT. The observed outputs form the monitored trace σ_{moni}. If a virtual clock is implemented in the IUT, time can be fast-forwarded to highly accelerate the IUT execution.

During execution, the input trace σ_{in} is stimulated on the IUT. The monitored traces σ_{moni} are *physical* output traces produced by the adaptor. As depicted in Figure 4.7, the adaptor has a main role during execution and is designed to modify as little as possible the IUT. If the IUT implements a virtual clock, time can be elapsed in a fast-forward mode, *i.e.* notifying to the clock that time is passed instead of waiting for it. It accelerates the execution, but in a way that falsify the context of time in the sense that the notion of waiting for an amount of time is not strictly tested in the IUT. However, it permits to execute long and huge benchmarks of tests very quickly.

Adaptors are related to the IUT, therefore, they cannot be implemented in a generic framework. The following interface describes a pattern that exists for such adaptors:

- $\mathsf{stimulate}(e)$, sends an event e to the IUT,
- $\mathsf{observe}(a)$, detects an action a sent at a time t by the IUT and
- $\mathsf{elapse}(d)$, optionally sends an amount of time to advance d^{phy} in seconds.

4.1.6 Comparison

Comparison returns whether the IUT conforms its model. Following the rtioco_e conformance defined in Equation 2.1, comparison concretizes the implementation relation which compares the two output traces: σ_{ref} and σ_{moni}. This relation is defined in Section 3.2 with the output trace equivalence $\cong$ of Equation 3.1. In other words, our framework follows rtioco_e conformance but comparing actions of same logical instant.

Notice that the rtioco_e conformance allows the system to make some divergences for other inputs. Indeed, it considers an implementation conform, if it follows at least the specified actions according to the specified inputs, supposing the behaviors after not specified inputs conform. However, because we consider the same input trace σ_{in} for the model and the IUT, only the expected inputs are considered in the framework.
As depicted in Figure 4.8, the comparison is done modulo logical instants with a tolerance ϵ between two timestamps. This tolerance is due to the time precision that can be arbitrary high, so we have to fix a bound to clearly specify after how much duration a timed error is effectively raised.

Comparison Algorithm. Compare two output traces σ and σ' over the same output alphabet Act, with no clue on the order of their actions, results

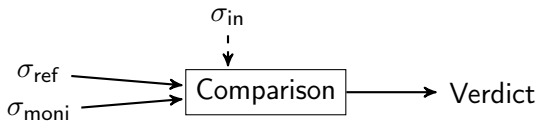

Figure 4.8: TMBT framework - comparison step. Comparison renders a verdict answering either the monitored trace σ_{moni} *passes* the test case or not. Given the reference trace σ_{ref}, the logical trace equivalence $\cong$ defined in Equation 3.1 is implemented. Comparison manages a tolerance ϵ between two timestamps.

in a complexity of $\mathcal{O}(m \times n)$ where m is the length of σ and n the length of σ'. The difficulty is that actions are ordered by their timestamp t, but each action can have a different t and may even be missed in one trace. Hence, the worst case is seen when the algorithm parses n actions for every m. The idea in our algorithm is to link the actions of the two output traces. First, the trace σ' is parsed constructing a map with its actions, denoted a'. Then, each action of σ, denoted a, is related to one a' or marked as unexpected. Finally, the comparison is done with another traversal of the two traces.

Hence, the algorithm traverses twice the trace σ', one to construct the map, the other to compare. Similarly, it traverses twice the trace σ, one to link the actions together, the other to compare. This algorithm computes and prints the verdict in $\mathcal{O}(2m + 2n)$ in time, and $\mathcal{O}(m + 2n)$ in space, because the map has the length of σ'.

More precisely, we let $a = \langle l, t, \alpha \rangle$ be an action of the output trace σ composed of a label l, a physical timestamp t and α a link to the related action a' if present in the trace σ', $\perp$ otherwise. Actions $a' = \langle l', t', \alpha' \rangle$ have the same form as actions a. Notice that we assume the labels known, and if different, the algorithm needs a symbol table to link l and l'. Comparison is performed as follows:

(1) We create the map $\mathsf{H} : l \to \langle l', t', \perp \rangle$ for each action a' of σ'. The map returns in $\mathcal{O}(1)$ the action a' related to the label l or $\perp$ if not present.

(2) $\forall a = \langle l, t, \perp \rangle \in \sigma$:

- $a = \langle l, t, a' \rangle$ and $a' = \langle l', t', a \rangle$, if $\mathsf{H}(l) = \langle l', t', \perp \rangle$,
- $a = \langle l, t, \perp \rangle$ otherwise.

(3) Finally, the Algorithm 1 is performed.

We initialize the map H with the actions a'. This map is used to assign the

related actions a and a' in the two traces and avoids to search actions later in the rest of the algorithm. The algorithm can access to the related action via the third field α.

Algorithm 1 Delta comparison algorithm.

Input: Two (linked) traces σ and σ', a tolerance ϵ.
Output: The verdict of $\sigma \cong \sigma'$.

```
function DeltaCompare(σ, σ', ε)
    ⟨l_i, t_i, α_i⟩ ← ⟨l_0, t_0, α_0⟩::σ
    ⟨l'_i, t'_i, α'_i⟩ ← ⟨l'_0, t'_0, α'_0⟩::σ'
    while a_i ≠ ∅ and a'_i ≠ ∅ do
        while |t_i − t'_i| ≤ ε do
            if ⟨l_i, t_i, ⟨l'_j, t'_j, a_i⟩⟩ and |t_i − t'_j| > ε then
                return fail
            else if ⟨l_i, t_i, ⊥⟩ then
                return fail
            end if
            a_i ← a_{i+1}::σ
        end while
        while |t_i − t'_i| > ε do
            if ⟨l'_i, t'_i, ⊥⟩ then
                return fail
            end if
            a'_i ← a'_{i+1}::σ'
        end while
    end while
    return pass
end function
```

Briefly, Algorithm 1 is a comparison of σ and σ' modulo logical instant. The current actions in σ and σ' denoted a_i and a'_i are triples $\langle l_i, t_i, \alpha_i \rangle$. First, they are initialized with the first action of each trace. Then, the algorithm loops until the two traces are handled.

The first loop compares the actions a of the trace σ while its timestamp t is in the current logical instant t'. The second loop checks the action a' of σ' until σ and σ' are in the same logical instant t again. In this second loop, we only check either α' is $\perp$. The timestamps comparison is useless because it is checked in the first loop, however, action in σ but not in σ' are not detected in this first step.

The algorithm can return a fail verdict in three cases:

1. if the timestamp of the related action $\langle l_j, t_j, a_i \rangle$ is different than t_i,
2. when the action $\langle l_i, t_i, \bot \rangle$ is not related, the action is called unexpected by the model, or,
3. when the specified action $\langle l'_i, t'_i, \bot \rangle$ is not in σ, this action is called missed by the IUT.

If no fail is returned, the algorithm returns a verdict pass.

Example 4.1.19: With such an algorithm, the following traces σ_1 and σ_2 are not equivalent but σ_1 and σ_3 are.

$$\sigma_1 = \langle a_1, 0 \rangle \cdot \langle a_2, 0 \rangle \cdot \langle a_3, 0.5 \rangle \cdot \langle a_5, 1 \rangle$$
$$\sigma_2 = \langle a_2, 0 \rangle \cdot \langle a_3, 0.5 \rangle \cdot \langle a_5, 1 \rangle$$
$$\sigma_3 = \langle a_2, 0 \rangle \cdot \langle a_1, 0 \rangle \cdot \langle a_3, 0.5 \rangle \cdot \langle a_5, 1 \rangle$$

Indeed, σ_2 misses the action a_1 and is not conformed to σ_1. However, σ_3 just switches actions a_1 and a_2. It is conformed to σ_1 because these two actions are in the same logical instant 0. ◊

4.1.7 Verdict

Our TMBT framework returns a verdict, which is a pretty printed document to inform testers of the IUT conformance *wrt.* its model. The document is spilt in logical instants in order to visualize clearly the actions related to an external event reception. Each action symbol $a \in \sigma_{\mathsf{moni}}$ is listed with its related model symbol or depicted as unexpected. A delta is depicted if more than ϵ duration is seen between the timestamps of the action and its related model action. Logical instants are separated by a duration in order to precise the time elapsed between two logical timestamps.

If the high-level requirements specify an ideal input trace, the difference between the ideal trace and the input trace σ_{in} can be depicted in the verdict. It permits to include events and their differences from the ideal input trace. This information improves the understanding of the verdict because one action can be managed in another manner if an event is early or late.

4.2 Model Construction Rules

Our TMBT framework has a construction phase, which, given a AST in input and a set of construction rules, builds automatically a IRTM. This section

formally defines the construction rules and details the overview done in Section 4.1.2. We introduce two operators to compose FSMs in Section 4.2.1 and present the preliminary and $\mathcal{M}_{\mathsf{env}}$ construction rules in Section 4.2.2 and Section 4.2.3 respectively. Then, we present several examples in order to apply these construction rules to build different system specifications $\mathcal{M}_{\mathsf{sys}}$ in Section 4.2.4.

The principle of construction rules is to define FSM related to items of the abstract syntax grammar. Moreover, the composition of these FSM is formally defined too. This construction is recursive, following the nested items defined by the grammar, hence, the rules have the form of an inference tree, constructing or composing FSMs. We define the composition of FSM by distinguishing two sequences of locations in the FSM definition: *providers* and *seekers*.

Definition 4.1. *A type of FSM is a pair $\langle p, s\rangle$ where p is its number of providers and s its number of seekers.*

In the following of the section, we shall sometimes write the type of FSM in exponent, as $\mathcal{A}^{\langle p,s\rangle}$, in order to make it explicit when needed. We call a FSM of type $\langle 0, 0\rangle$ *complete* and depict, in the graphical representation of FSMs, provider and seeker locations with $i\circ$ and $\prec i$ where i is an index in the sequence of providers, resp. seekers.

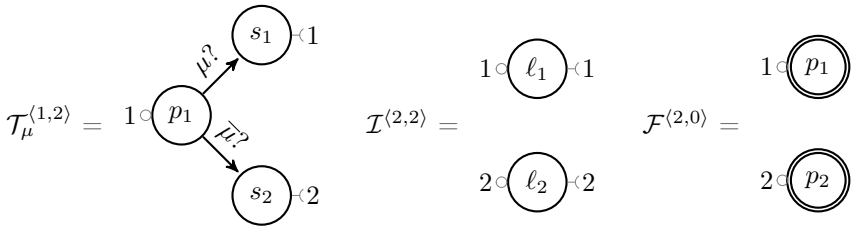

Figure 4.9: The most used FSMs: $\mathcal{T}^{\langle 1,2\rangle}$, $\mathcal{I}^{\langle 2,2\rangle}$ and $\mathcal{F}^{\langle 2,0\rangle}$.

Example 4.2.20: We define three FSMs depicted in Figure 4.9 commonly used during the construction:

- triggers $\mathcal{T}_{\mu}^{\langle 1,2\rangle}$, with $\mu \in \mathit{Evt} \cup \mathit{Sig}$, for starting a FSM at the detection of some input symbol μ. Every trigger FSM has one provider and two seekers, corresponding to a start of the FSM in a normal or an error *mode*,
- idlers $\mathcal{I}^{\langle i,i\rangle}$, where each location is both a provider and a seeker and

- enders $\mathcal{F}^{\langle i,0\rangle}$, with an empty list of seekers.

$\Diamond$

4.2.1 Operators

The parallel and sequential concatenation operators are denoted by $\|$ and $+$ respectively. Both of them take two FSMs and return one FSM composed of their two arguments. The operators are defined according to the type of FSM they manage.

The binary operator $\| : \langle n, m\rangle \to \langle n', l\rangle \to \langle p, s\rangle$, with $p = max(n, n')$ and $s = m + l$, is a parallel composition of FSMs defined as follows. Let $\mathcal{A} = \langle \Sigma^{\mathcal{A}}_{\mathsf{in}}, \Sigma^{\mathcal{A}}_{\mathsf{out}}, \mathcal{L}^{\mathcal{A}}, \ell^{\mathcal{A}}_0, \Delta^{\mathcal{A}}\rangle$ and $\mathcal{B} = \langle \Sigma^{\mathcal{B}}_{\mathsf{in}}, \Sigma^{\mathcal{B}}_{\mathsf{out}}, \mathcal{L}^{\mathcal{B}}, \ell^{\mathcal{B}}_0, \Delta^{\mathcal{B}}\rangle$, be two FSMs of respective types $\langle n, m\rangle$ and $\langle n', l\rangle$ and with respective sequences of providers and seekers: $p^{\mathcal{A}}_1, \ldots, p^{\mathcal{A}}_n$, and $s^{\mathcal{A}}_1, \ldots, s^{\mathcal{A}}_m \in \mathcal{L}^{\mathcal{A}}$, $p^{\mathcal{B}}_1, \ldots, p^{\mathcal{B}}_{n'}$, and $s^{\mathcal{B}}_1, \ldots, s^{\mathcal{B}}_l \in \mathcal{L}^{\mathcal{B}}$, with $\mathcal{L}^{\mathcal{A}}$ and $\mathcal{L}^{\mathcal{B}}$ disjoint. Their parallel composition is defined by

$$\langle \Sigma^{\mathcal{A}}_{\mathsf{in}} \cup \Sigma^{\mathcal{B}}_{\mathsf{in}}, \Sigma^{\mathcal{A}}_{\mathsf{out}} \cup \Sigma^{\mathcal{B}}_{\mathsf{out}}, \mathcal{L}^{\mathcal{A}} \mathbin{\mathring{\cup}} \mathcal{L}^{\mathcal{B}} \mathbin{\mathring{\cup}} \{\ell_0, \ldots, \ell_p\}, \ell_0, \Delta^{\mathcal{A}} \mathbin{\mathring{\cup}} \Delta^{\mathcal{B}} \mathbin{\mathring{\cup}} \Delta\rangle$$

where:

- $\ell_0, \ldots, \ell_p$ are new locations,
- the sequences of providers and seekers of $\mathcal{A}\|\mathcal{B}$ are respectively $\ell_0, \ldots, \ell_p$ and $s^{\mathcal{A}}_1, \ldots, s^{\mathcal{A}}_m, s^{\mathcal{B}}_1, \ldots, s^{\mathcal{B}}_l$ and
- Δ contains the set of transitions of the form $\ell_i \xrightarrow{and} \ell^{\mathcal{A}}_{i_1} \| \ell^{\mathcal{B}}_{i_2}$, with $1 \le i \le p$, such that if $i \le n$ then $i_1 = i$, otherwise $i_1 = n$, and if $i \le n'$ then $i_2 = i$, otherwise $i_2 = n'$.

Example 4.2.21: Figure 4.10 depicts an example of parallel composition of two FSMs. Note that the set of input and output symbols of $\mathcal{A}$ and $\mathcal{B}$ are not required to be disjoint since these symbols are used for communication between the two FSMs after composition. The FSM $\mathcal{A}$ on the left has one provider and one seeker. On the right, the FSM $\mathcal{B}$ has two providers and seekers. During composition, two *and*-transitions are created according to the two providers of $\mathcal{B}$. FSMs $\mathcal{A}$ and $\mathcal{B}$ are composed by setting the first providers p^A_1 with p^B_1 in parallel with the first *and*-transition, and p^A_1 and p^B_2 with the second. After composition, the FSM $\mathcal{C}$ launches the FSM $\mathcal{A}$ both in locations p_1 and p_2, however, only the corresponding provider is continued for FSM $\mathcal{B}$. The location ℓ_2 for p_1 and ℓ_3 for p_2. $\Diamond$

The binary operator $+ : \langle k, n\rangle \to \langle n', m\rangle \to \langle k, m\rangle$, is a sequential composition of FSMs. Let $\mathcal{A} = \langle \Sigma^{\mathcal{A}}_{\mathsf{in}}, \Sigma^{\mathcal{A}}_{\mathsf{out}}, \mathcal{L}^{\mathcal{A}}, \ell^{\mathcal{A}}_0, \Delta^{\mathcal{A}}\rangle$ and $\mathcal{B} = \langle \Sigma^{\mathcal{B}}_{\mathsf{in}}, \Sigma^{\mathcal{B}}_{\mathsf{out}}, \mathcal{L}^{\mathcal{B}}, \ell^{\mathcal{B}}_0, \Delta^{\mathcal{B}}\rangle$,

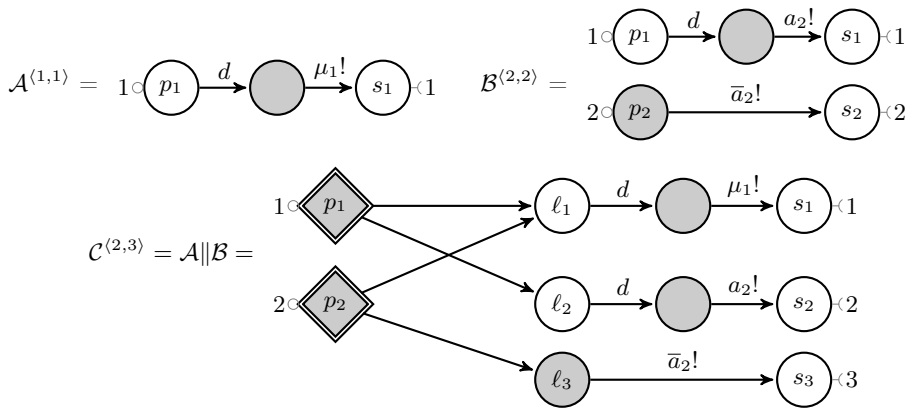

Figure 4.10: Parallel composition of two FSMs.

be two FSMs with respective types $\langle k, n\rangle$ and $\langle n', m\rangle$ and with respective sequences of providers and seekers: $p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}$, and $s_1^{\mathcal{A}}, \ldots, s_n^{\mathcal{A}} \in \mathcal{L}^{\mathcal{A}}$, $p_1^{\mathcal{B}}, \ldots, p_{n'}^{\mathcal{B}}$, and $s_1^{\mathcal{B}}, \ldots, s_m^{\mathcal{B}} \in \mathcal{L}^{\mathcal{B}}$. Their sequential composition is defined by

$$\langle \Sigma_{\mathsf{in}}^{\mathcal{A}} \cup \Sigma_{\mathsf{in}}^{\mathcal{B}}, \Sigma_{\mathsf{out}}^{\mathcal{A}} \cup \Sigma_{\mathsf{out}}^{\mathcal{B}}, \mathcal{L}', \ell_0', \Delta\rangle$$

with $\mathcal{L}' = (\mathcal{L}^{\mathcal{A}} \setminus \{s_1^{\mathcal{A}}, \ldots, s_n^{\mathcal{A}}\}) \mathbin{\mathring{\cup}} (\mathcal{L}^{\mathcal{B}} \setminus \{p_1^{\mathcal{B}}, \ldots, p_{n'}^{\mathcal{B}}\}) \mathbin{\mathring{\cup}} \{\ell_1, \ldots, \ell_{n''}\}$. Where:

- $\ell_1, \ldots, \ell_{n''}$ are new locations, not in $\mathcal{L}^{\mathcal{A}} \cup \mathcal{L}^{\mathcal{B}}$, and $n'' = min(n, n')$,
- the sequences of providers and seekers of $\mathcal{A}+\mathcal{B}$ are respectively $p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}$ and $s_1^{\mathcal{B}}, \ldots, s_m^{\mathcal{B}}$,
- $\ell_0' = \ell_i$ if there exists $i \leq n$ such that $\ell_0^{\mathcal{A}} = s_i^{\mathcal{A}}$, otherwise $\ell_0' = \ell_0^{\mathcal{A}}$ and finally
- the set of transitions Δ is obtained by replacing in $\Delta^{\mathcal{A}} \cup \Delta^{\mathcal{B}}$ every location $s_i^{\mathcal{A}}$ or $p_i^{\mathcal{B}}$ by ℓ_i for $1 \leq i \leq n''$.

Intuitively, every seeker of $\mathcal{A}$ is merged with the provider of $\mathcal{B}$ with the same index. Note that if $n > n'$, then the seekers $s_{n'+1}^{\mathcal{A}}, \ldots, s_n^{\mathcal{A}}$ of $\mathcal{A}$ without matching providers in $\mathcal{B}$ become *exit* locations in $\mathcal{A}+\mathcal{B}$. If $n < n'$, then the providers $p_{n+1}^{\mathcal{B}}, \ldots, p_{n'}^{\mathcal{B}}$ of $\mathcal{B}$ without matching seekers in $\mathcal{A}$ are deleted and become standard locations in $\mathcal{A}+\mathcal{B}$ (neither providers nor seekers).

Notice that we define later special connectors. These connectors are not handled by the parallel concatenation operator. However, they are handled as usual connectors in sequential operators.

Example 4.2.22: Figure 4.11 depicts an example of sequential composition of two FSMs. The resulted FSM $\mathcal{C}$ begins with FSM $\mathcal{A}$ and continues with FSM $\mathcal{B}$, merging the seekers of $\mathcal{A}$ with the providers of $\mathcal{B}$. ◊

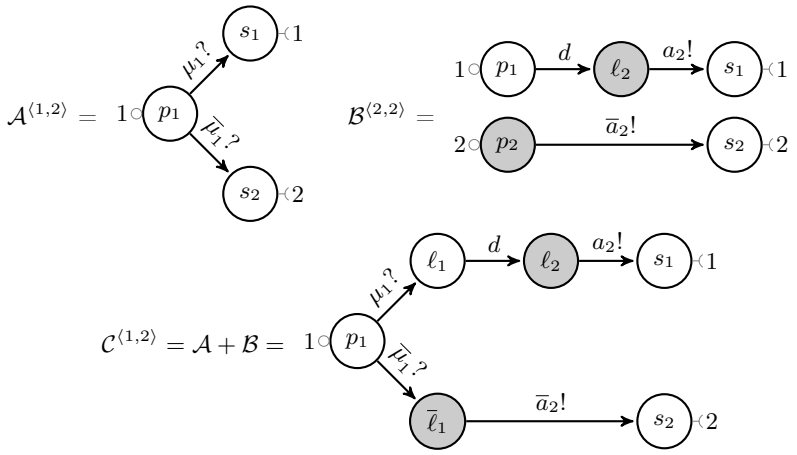

Figure 4.11: Sequential composition of two FSMs.

4.2.2 Rules for requirements

We introduce the construction rules in this section. For our examples, we define the rules over the grammar depicted in Figure 4.12.

requirement	::=	ε | event requirement	
event	::=	evt(e, duration, sequence)	$e \in Evt$
sequence	::=	ε | action sequence	
action	::=	act(duration, sequence) | act(duration, a)	$a \in Act$
duration	::=	d timeUnit	$d \in \mathbb{R}_*^+$
timeUnit	::=	s | t_1	

Figure 4.12: Example of grammar followed by ASTs in input.

Inference rules. The construction is defined using rules of the following form:

$$\langle aux \rangle : \langle \mathsf{AST} \rangle \vdash_{rule} \langle \mathsf{FSM} \rangle$$

where $\langle aux \rangle$ is a sequence of auxiliary arguments, $\langle \mathsf{AST} \rangle$ is the element of the parsed requirements (in abstract syntax) and $\langle \mathsf{FSM} \rangle$ is the corresponding FSM constructed and returned.

Processing of requirement. The rule $\vdash_{\mathsf{all}}$ constructs the FSM associated to the parsed requirements.

$$\frac{}{: \emptyset \vdash_{\mathsf{all}} \mathcal{A}_\emptyset} \qquad \frac{: req \vdash_{\mathsf{env}} \mathcal{M}_{\mathsf{env}} \qquad : req \vdash_{\mathsf{sys}_1} \mathcal{A}_1 \qquad \ldots \qquad : req \vdash_{\mathsf{sys}_n} \mathcal{A}_n}{: req \vdash_{\mathsf{all}} \mathcal{M}_{\mathsf{env}} \| \mathcal{A}_1 \| \ldots \| \mathcal{A}_n}$$

If the AST is empty, the rule $\vdash_{\mathsf{all}}$ returns an empty FSM $\mathcal{A}_\emptyset$, a FSM with an empty set of locations. Otherwise, $n+1$ rules are applied to the requirements req, with n, the number of rules composing the specification $\mathcal{M}_{\mathsf{sys}}$. Among this rules, $\vdash_{\mathsf{env}}$ constructs the environment model $\mathcal{M}_{\mathsf{env}}$.

Hence we define $\mathcal{M} = \mathcal{M}_{\mathsf{env}} \| \mathcal{M}_{\mathsf{sys}}$ with $\mathcal{M}_{\mathsf{sys}} = \mathcal{A}_1 \| \ldots \| \mathcal{A}_n$: the model $\mathcal{M}$ is the result of the parallelization of the environment model $\mathcal{M}_{\mathsf{env}}$ and the specification $\mathcal{M}_{\mathsf{sys}}$. The specification is itself the parallel composition of the other n models. Notice that these FSMs must have the type $\langle 1, 0 \rangle$ and that $\mathcal{M}$ is complete.

4.2.3 Environment rules

Formally, $\mathcal{M}_{\mathsf{env}}$ is a non-deterministic FSM of the form $\langle Act, Evt, \mathcal{L}^{\mathcal{E}}, \ell_0^{\mathcal{E}}, \Delta^{\mathcal{E}} \rangle$, where the partition of the output alphabet $\Sigma_{\mathsf{out}} = Evt$ is $\Sigma_{\mathsf{out}}^{\mathsf{sig}} = Evt$ and $\Sigma_{\mathsf{out}}^{\mathsf{ext}} = \emptyset$. An important point when dealing with a IRTM is that action emissions are never restricted because action receptions are not explicit in the test environment. Hence, $\mathcal{M}_{\mathsf{env}}$ is always input-enabled and can accept all its input symbols in every location.

One can construct several test environments to represent different test hypotheses. In the following of the section, we present two examples of test environment construction. First, we construct a generic environment which is easy to specify. Then, we present an environment musically relevant with parameters to restrict the possible input traces of the test. The two environment FSMs are built in a single pass through the AST.

1) Generic Environment

For the simple case, we want to construct a generic environment. Such an environment can send all events in Evt at anytime. Remark that to simplify the construction here, we do not manage event durations. We specify the rule $\vdash_{\mathsf{env}}$ to parse the sequence of events until the empty list, handling each event evt of the requirements req:

$$\frac{: \mathsf{evt}(e, d, as) \vdash_{\mathsf{evt}} \mathcal{E}^{\langle 1,1 \rangle} \qquad : req' \vdash_{\mathsf{env}} \mathcal{E}^{\langle 1,0 \rangle}}{: \mathsf{evt}(e, d, as) :: req' \vdash_{\mathsf{env}} \mathcal{E}^{\langle 1,1 \rangle} + \mathcal{E}^{\langle 1,0 \rangle}} \qquad \frac{}{: \emptyset \vdash_{\mathsf{env}} \mathcal{F}^{\langle 1,0 \rangle}}$$

The rule $\vdash_{\mathsf{evt}}$ handles each event of the AST by adding a possible emission of the corresponding event symbol e to the unique location of the FSM. The FSMs constructed by the rules $\vdash_{\mathsf{evt}}$ and $\vdash_{\mathsf{env}}$ are depicted in Figure 4.13. Then, the rule $\vdash_{\mathsf{env}}$ assembles the FSMs $\mathcal{E}^{\langle 1,1\rangle}$ and $\mathcal{E}^{\langle 1,0\rangle}$. These FSMs are constructed for the current event and the next events constructed by the recursive call of $\vdash_{\mathsf{env}}$ with the rest of the requirements req' respectively. At the end, for the empty event sequence case, an ender FSM terminates the construction of $\mathcal{M}_{\mathsf{env}}$. Finally, $\mathcal{M}_{\mathsf{env}}$ can send all the events in Evt.

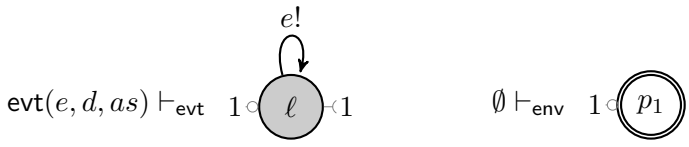

Figure 4.13: Parts of the generic environment FSM.

2) Musical Environment

For the second example, we detail the construction of a test environment relevant in a musical context, and refer to the high-level requirements as the score. We assume an ordered sequence of events as specified in the given score and for each event, an event duration to wait before sending the next event. Hence, the input sequence defined by this environment model must be of the form: $e_1!\cdot d_1^e\cdot\ldots\cdot d_{n-1}^e\cdot e_n!\cdot d_n^e$ with $e_i \in Evt$ and d_i^e the duration of event e_i. We add several options to construct the model $\mathcal{M}_{\mathsf{env}}$:

- n_{err}, the number of consecutive events possibly missing and
- κ, the percentage of variation tolerated on the event's durations.

This second parameter permits the creation of bounds of the form $[d_i^e(1-\kappa), d_i^e(1+\kappa)]$ centered around the duration d_i^e specified in the score, called the ideal duration. We present the construction of $\mathcal{M}_{\mathsf{env}}$ for the value of $n_{err} = 1$.

$$\frac{: \mathsf{evt}(e,d,as) \vdash_{\mathsf{evt}_0} \mathcal{E}_0^{\langle 1,2\rangle} \qquad : req' \vdash_{\mathsf{env}_1} \mathcal{E}^{\langle 2,0\rangle}}{: \mathsf{evt}(e,d,as)::req' \vdash_{\mathsf{env}} \mathcal{E}_0^{\langle 1,2\rangle} + \mathcal{E}^{\langle 2,0\rangle}}$$

$$\frac{: \mathsf{evt}(e,d,as) \vdash_{\mathsf{evt}_1} \mathcal{E}_1^{\langle 2,2\rangle} \qquad : req' \vdash_{\mathsf{env}_1} \mathcal{E}^{\langle 2,0\rangle}}{: \mathsf{evt}(e,d,as)::req' \vdash_{\mathsf{env}_1} \mathcal{E}_1^{\langle 2,2\rangle} + \mathcal{E}^{\langle 2,0\rangle}} \qquad \frac{}{: \emptyset \vdash_{\mathsf{env}_1} \mathcal{F}^{\langle 2,0\rangle}}$$

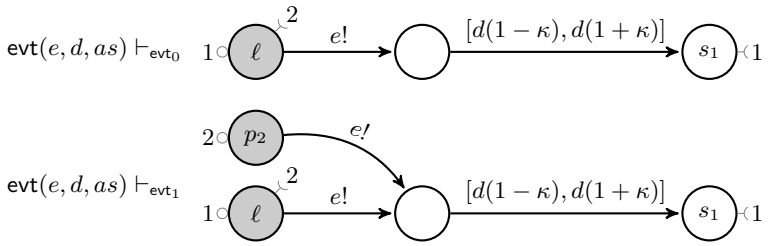

Figure 4.14: Parts of the musical environment FSM.

The rule $\vdash_{\mathsf{evt}_0}$, on the top, initializes the FSM $\mathcal{M}_{\mathsf{env}}$, by emitting the first event $\mathsf{evt}(e, d, s)$ and waiting for a duration in the interval $[d(1-\kappa), d(1+\kappa)]$. The rule $\vdash_{\mathsf{evt}_1}$, on the bottom, treats each following event of the requirements, with the possibility to emit the current event e from the previous step (provider 1) or the step before (provider 2). The FSMs constructed by these two rules are depicted in Figure 4.14. Notice that in this case, we added the rule $\vdash_{\mathsf{evt}_0}$ to the set of rules for handling the first event, because we cannot miss an earlier event than the first one. Then, the rules $\vdash_{\mathsf{env}}$ and $\vdash_{\mathsf{env}_1}$, assemble the FSMs for the first, respectively next, events. At the end, the same ender FSM is constructed for the empty sequence case but with two seekers.

$\mathsf{evt}(e_1, 1s, as_1);$
$\mathsf{evt}(e_2, 1s, as_2);$ where
$as_1 = \mathsf{act}(1s, [a_1])$
$as_2 = \mathsf{act}(0.5s, [as_3]);\ \mathsf{act}(0.5s, [a_2])$
$as_3 = \mathsf{act}(1s, [a_3])$

Figure 4.15: An abstract syntax tree example.

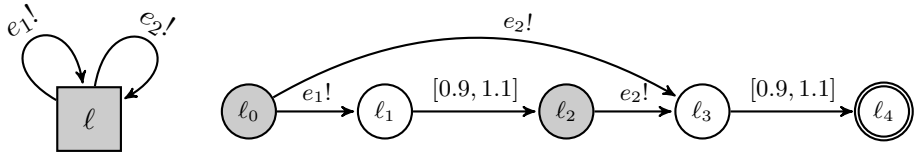

Figure 4.16: Two environment models constructed from the same inputs. On the left, the generic model rules, on the right the musical relevant ones with $\kappa = 0.10$ and $n_{err} = 1$.

Example 4.2.23: Figure 4.16 depicts the two $\mathcal{M}_{\mathsf{env}}$ models constructed from the applications of the two environment rules examples to the AST depicted in Figure 4.15. The requirements specify the emission of an action a_i after each event e_i lasting 1 second for $i \in \{1, 2\}$. The generic environment model, on the left, allows to send these events at every moment. The second musical environment, applied with $\kappa = 0.10$ and $n_{err} = 1$, allows to emit e_1 or e_2. However:

1. in case of e_1 first, waits for a duration between 0.9 and 1.1 seconds before emitting e_2 and prevents from sending e_1 twice, and,

2. in case of e_2 emitted first (constructed from the index 2 of the rule $\vdash_{\mathsf{evt}_1}$), stops emitting events.

$\Diamond$

4.2.4 Toy example model

We continue the presentation of our construction rules with respect to the abstract syntax depicted in Figure 4.12. In this section, we detail three examples to construct the IUT specification $\mathcal{M}_{\mathsf{sys}}$ according to different approaches.

Our specification aims at constructing models expressing requirements such as: *I want action a_1 after each event e_1 or 3 seconds after e_2*. There are lots of systems that can be targeted by such specifications. We can of course cite IMS as MAX-MSP [78] and Chuck [48] systems but can include home-automation, robotics or internet of things systems.

The abstract syntax of Section 4.1.1 can express this kind of specifications. Indeed, the syntax allows to specify an action sequence *as* triggered by either an event or an amount of time after an event detection or another action activation. The three next examples are based on this syntax and aim at constructing such kind of specifications. The examples are three construction options according to different semantics of the abstract syntax.

First, we present the main rule as follows:

$$\frac{: req \vdash_{\mathsf{env}} \mathcal{M}_{\mathsf{env}} \qquad : req \vdash_{\mathsf{sys}} \mathcal{M}_{\mathsf{sys}}}{: req \vdash_{\mathsf{all}} \mathcal{M}_{\mathsf{env}} \| \mathcal{M}_{\mathsf{sys}}}$$

The rule $\vdash_{\mathsf{sys}}$ constructs the IUT specification $\mathcal{M}_{\mathsf{sys}}$ aside of test envirenment $\mathcal{M}_{\mathsf{env}}$.

1) Event reaction

For the first example, the IUT must react to event detections but only once for each symbol. The only rule $\vdash_{\mathsf{sys}}$, managing the FSMs of $\mathcal{M}_{\mathsf{sys}}$, constructs a FSM of the form $\mathcal{A} = \langle Evt, Act, \mathcal{L}^{\mathcal{A}}, \ell_0^{\mathcal{A}}, \Delta^{\mathcal{A}} \rangle$ specifying behaviors of the IUT in reaction to the events of the test environment.

$$\frac{e : as \vdash_{\mathsf{seq}} \mathcal{A}_s^{\langle 1,0\rangle} \qquad : req' \vdash_{\mathsf{sys}} \mathcal{A}^{\langle 1,0\rangle}}{: \mathsf{evt}(e,d,as)::req' \vdash_{\mathsf{sys}} \mathcal{A}_s^{\langle 1,0\rangle} \| \mathcal{A}^{\langle 1,0\rangle}} \qquad \frac{: req' \vdash_{\mathsf{sys}} \mathcal{A}^{\langle 1,0\rangle}}{: \mathsf{evt}(e,d,\varepsilon)::req' \vdash_{\mathsf{sys}} \mathcal{A}^{\langle 1,0\rangle}}$$

$$\frac{}{: \emptyset \vdash_{\mathsf{sys}} \mathcal{F}^{\langle 1,0\rangle}}$$

The FSM $\mathcal{A}_s$ of type $\langle 1,0\rangle$ is associated to $\mathsf{evt}(e,d,as)$ and describes the behavior of an action sequence as triggered by an event e. The rule $\vdash_{\mathsf{sys}}$ returns the parallel composition of $\mathcal{A}_s$ with the FSM $\mathcal{A}$ built by a recursive call of $\vdash_{\mathsf{sys}}$ with the rest of the requirements req'. On the right, the second rule $\vdash_{\mathsf{sys}}$ returns the result of its recursive call uniquely. This second case happens when an event does not trigger action (denoted with ε in the third field of evt). When the end of requirements is reached, the FSM is terminated by adding an ender $\mathcal{F}^{\langle 1,0\rangle}$ with 1 provider.

We now define the rule $\vdash_{\mathsf{seq}}$ for building the FSMs associated to an action sequence as. The rule parses a sequence of actions as and sends these actions accordingly.

$$\frac{e : \vdash_{\mathsf{seq}_0} \mathcal{A}_e^{\langle 1,1\rangle} \qquad : as \vdash_{\mathsf{seq}_1} \mathcal{A}^{\langle 1,0\rangle}}{e : as \vdash_{\mathsf{seq}} \mathcal{A}_e^{\langle 1,1\rangle} + \mathcal{A}^{\langle 1,0\rangle}}$$

$$\frac{d : \vdash_{\mathsf{delay}} \mathcal{A}_d^{\langle 1,1\rangle} \qquad : a : \vdash_{\mathsf{act}} \mathcal{A}_a^{\langle 1,1\rangle} \qquad : as' \vdash_{\mathsf{seq}_1} \mathcal{A}^{\langle 1,0\rangle}}{: \mathsf{act}(d,a)::as' \vdash_{\mathsf{seq}_1} \mathcal{A}_d^{\langle 1,1\rangle} + \mathcal{A}_a^{\langle 1,1\rangle} + \mathcal{A}^{\langle 1,0\rangle}}$$

$$\frac{d : \vdash_{\mathsf{delay}} \mathcal{A}_d^{\langle 1,1\rangle} \qquad : as_a \vdash_{\mathsf{seq}_1} \mathcal{A}_s^{\langle 1,0\rangle} \qquad : as' \vdash_{\mathsf{seq}_1} \mathcal{A}^{\langle 1,0\rangle}}{: \mathsf{act}(d,as_a)::as' \vdash_{\mathsf{seq}_1} \mathcal{A}_d^{\langle 1,1\rangle} + (\mathcal{A}_s^{\langle 1,0\rangle} \| \mathcal{A}^{\langle 1,0\rangle})} \qquad \frac{}{: \emptyset \vdash_{\mathsf{seq}_1} \mathcal{F}^{\langle 1,0\rangle}}$$

The rule $\vdash_{\mathsf{seq}}$, on the top, initializes the FSM, by receiving the event e given in the rule argument and applying the rule $\vdash_{\mathsf{seq}_1}$ to the sequence as. The rule $\vdash_{\mathsf{seq}_1}$ treats each following action of as. Three cases can be constructed according to the item parsed:

(1) An atomic action $\mathsf{act}(d, a)$ is parsed (middle rule): The rule $\vdash_{\mathsf{delay}}$ waits for the duration d if greater than 0, and the rule $\vdash_{\mathsf{act}}$ sends the action symbol a. The FSMs are sequentially concatenated with $\mathcal{A}$, built by the recursive call of $\vdash_{\mathsf{seq}_1}$ with as'.

(2) A sub-sequence of actions $\mathsf{act}(d, as_a)$ is parsed (bottom left): As for an atomic action, $\vdash_{\mathsf{delay}}$ waits for the duration d. The FSM is sequentially concatenated with $\mathcal{A}_s$ and $\mathcal{A}$ built by two recursive calls of $\vdash_{\mathsf{seq}_1}$ with the sub-sequence as_a and the rest of the sequence as' respectively.

(3) The empty sequence is reached (bottom right): an ender is added to the FSM with provider 1.

The FSMs constructed by these rules for the base cases are depicted in Figures 4.17 and 4.18. Notice that here, they return a *recv*-, a *wait*- and an *emit*- transition.

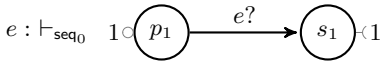

Figure 4.17: FSM constructed by $\vdash_{\mathsf{seq}_0}$.

Figure 4.18: FSMs constructed by $\vdash_{\mathsf{seq}_1}$.

Example 4.2.24: We show the model $\mathcal{M}$ obtained by the application of the construction rules above to the AST depicted in Figure 4.15. The requirements specify the emission of the action a_1, 1 second after the detection of the corresponding event e_1. Then, a_2 (resp. a_3) must be emitted 1 second (resp. 1.5 seconds) after e_2 detection. Notice that a_3 is triggered with a sub-sequence of actions as_3. The model $\mathcal{M}_{\mathsf{sys}}$ is depicted in Figure 4.19. We omit the test environment and denote it with $\mathcal{M}_{\mathsf{env}}$. The FSMs built at each application of a rules $\vdash_{\mathsf{seq}_0}$ and $\vdash_{\mathsf{seq}_1}$ are framed and annotated with: $\mathcal{A}_a^{\langle p,s\rangle}$ where $\langle p, s\rangle$ is the type and a is the AST item parsed, or, $\mathcal{F}$ when terminated. ◊

2) Loop reaction

In the second example, the IUT must react to several event detections when the sequence of actions related to this event symbol is terminated. This

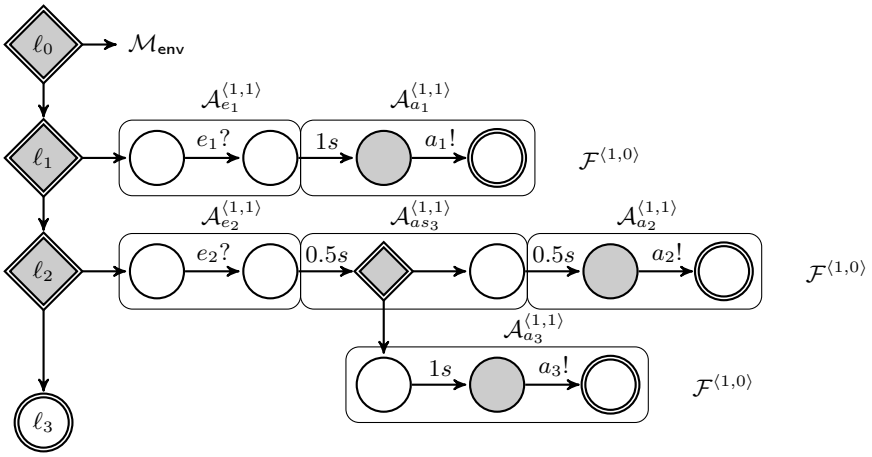

Figure 4.19: $\mathcal{M}_{\mathsf{sys}}$ resulted from the event reaction rules.

example begins with the same rule $\vdash_{\mathsf{sys}}$ than the first example, managing the FSMs of $\mathcal{M}_{\mathsf{sys}}$.

In order to specify loops in FSMs, we let ℓ be a special connector which can be both providers and seekers. This connector allows to specify a location stored in the continuation of the construction. Moreover, ℓ is not merged during parallel compositions of FSMs, but is concatenated with sequential composition operators. Such a connector is denoted as a pre-exponent ${}^{\ell}\mathcal{A}$ in a FSM. Moreover, we define a *linker* in order to merge connectors. A linker ${}^{\langle i,j\rangle}\mathcal{L}$ is a FSM of type $\langle 2, 0\rangle$ with two providers, i and j. It is the only way for composing a special connector with other connectors.

We now define the rule $\vdash_{\mathsf{seq}}$ for the second example. This rule builds the FSMs associated to an action sequence as according to the second approach. The rule parses a sequence of actions as and sends these actions accordingly.

$$\frac{e : \vdash_{\mathsf{seq}_0} {}^{\ell}\mathcal{A}_e^{\langle 1,1\rangle} \qquad : as \vdash_{\mathsf{seq}_2} {}^{\ell}\mathcal{A}^{\langle 1,0\rangle}}{e : as \vdash_{\mathsf{seq}} {}^{\ell}\mathcal{A}_e^{\langle 1,1\rangle} + {}^{\ell}\mathcal{A}^{\langle 1,0\rangle}}$$

$$\frac{d : \vdash_{\mathsf{delay}_2} {}^{\ell}\mathcal{A}_d^{\langle 1,1\rangle} \qquad : a : \vdash_{\mathsf{act}_2} {}^{\ell}\mathcal{A}_a^{\langle 1,1\rangle} \qquad : as' \vdash_{\mathsf{seq}_2} {}^{\ell}\mathcal{A}^{\langle 1,0\rangle}}{: \mathsf{act}(d,a)::as' \vdash_{\mathsf{seq}_2} {}^{\ell}\mathcal{A}_d^{\langle 1,1\rangle} + {}^{\ell}\mathcal{A}_a^{\langle 1,1\rangle} + {}^{\ell}\mathcal{A}^{\langle 1,0\rangle}}$$

$$\frac{d : \vdash_{\mathsf{delay}_2} {}^{\ell}\mathcal{A}_d^{\langle 1,1\rangle} \qquad : as_a \vdash_{\mathsf{seq}_1} \mathcal{A}_s^{\langle 1,0\rangle} \qquad : as' \vdash_{\mathsf{seq}_2} {}^{\ell}\mathcal{A}^{\langle 1,0\rangle}}{: \mathsf{act}(d,as_a)::as' \vdash_{\mathsf{seq}_2} {}^{\ell}\mathcal{A}_d^{\langle 1,1\rangle} + (\mathcal{A}_s^{\langle 1,0\rangle} \| {}^{\ell}\mathcal{A}^{\langle 1,0\rangle})}$$

$$\frac{}{: \emptyset \vdash_{\mathsf{seq}_2} {}^{\langle \ell,1\rangle}\mathcal{L}}$$

The rule $\vdash_{\mathsf{seq}}$, on the top, behaves as the first approach but adds a special seeker ℓ to the first location of the FSM and applies the rule $\vdash_{\mathsf{seq}_2}$ to as. The rule $\vdash_{\mathsf{seq}_2}$ treats similarly to $\vdash_{\mathsf{seq}_1}$ the parsed items, but stores the connector ℓ in addition. Notice that when a sub-sequence of actions $\mathsf{act}(d, as_a)$ is parsed, the previous rule $\vdash_{\mathsf{seq}_1}$ is applied to the sequence as_a. At the end, instead of terminating with an ender $\mathcal{F}^{\langle 1,0\rangle}$, a linker is constructed to merge the connector 1 with the special connector ℓ. The FSMs constructed by these rules are depicted in Figure 4.20.

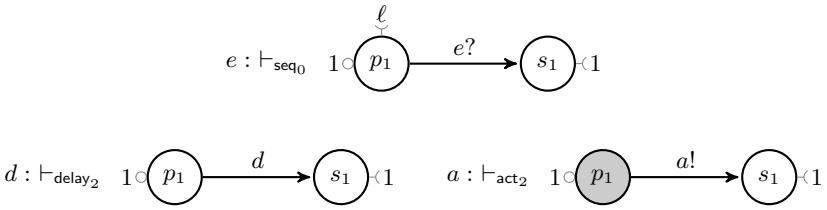

Figure 4.20: FSM constructed by $\vdash_{\mathsf{seq}_2}$ and the new $\vdash_{\mathsf{seq}_0}$.

Example 4.2.25: We show the model $\mathcal{M}$ obtained from the application of the second construction rules to the AST depicted in Figure 4.15. The model $\mathcal{M}_{\mathsf{sys}}$ is depicted in Figure 4.21.

The linkers FSM, constructed with seekers ℓ and 1, merge the last location with the first one, creating the required loop. Notice that the model constructed for as_3 is not modified, because the $\vdash_{\mathsf{seq}_1}$ rule is applied to the sub-sequence of actions as_3. ◊

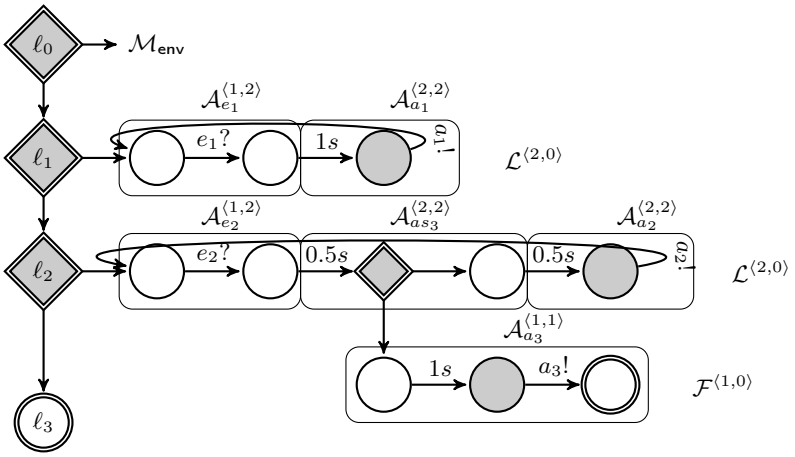

Figure 4.21: $\mathcal{M}_{\mathsf{sys}}$ resulted from the loop reaction rules.

3) Concurrent reaction

In the third example, the IUT must react to every event detection. However, to prevent from an unbound number of running FSMs, an event can be emitted to terminate all the FSMs related to its action sequences. The example begins with the same rule $\vdash_{\mathsf{sys}}$ than the first example, managing the FSMs of $\mathcal{M}_{\mathsf{sys}}$.

We add the events λ_1 and λ_2 in Evt, to terminate FSMs related to e_1 and e_2 respectively. Moreover, we add another special connector k to the existing ℓ. Finally, the event λ is given to $\vdash_{\mathsf{seq}}$ in parameters with its related event e.

We now define the rule $\vdash_{\mathsf{seq}}$ for the third example. This rule builds the FSMs associated to an action sequence as according to the third approach. The rule parses a sequence of actions as and sends these actions accordingly.

$$\frac{e, \lambda : \vdash_{\mathsf{seq}_0} {}^{\langle \ell,k \rangle}\mathcal{A}_e^{\langle 1,1 \rangle} \qquad \lambda : as \vdash_{\mathsf{seq}_3} {}^{k}\mathcal{A}^{\langle 1,0 \rangle}}{e, \lambda : as \vdash_{\mathsf{seq}} {}^{\langle \ell,k \rangle}\mathcal{A}_e^{\langle 1,1 \rangle} + ({}^{\langle \ell,1 \rangle}\mathcal{L} \| {}^{k}\mathcal{A}^{\langle 1,0 \rangle})}$$

$$\frac{d, \lambda : \vdash_{\mathsf{delay}_3} {}^{k}\mathcal{A}_d^{\langle 1,1 \rangle} \qquad a : \vdash_{\mathsf{act}_3} {}^{k}\mathcal{A}_a^{\langle 1,1 \rangle} \qquad \lambda : as' \vdash_{\mathsf{seq}_3} {}^{k}\mathcal{A}^{\langle 1,0 \rangle}}{\lambda : \mathsf{act}(d, a)::as' \vdash_{\mathsf{seq}_3} {}^{k}\mathcal{A}_d^{\langle 1,1 \rangle} + {}^{k}\mathcal{A}_a^{\langle 1,1 \rangle} + {}^{k}\mathcal{A}^{\langle 1,0 \rangle}}$$

$$\frac{d : \vdash_{\mathsf{delay}_3} {}^{k}\mathcal{A}_d^{\langle 1,1 \rangle} \qquad \lambda : as_a \vdash_{\mathsf{seq}_3} {}^{k}\mathcal{A}_s^{\langle 1,0 \rangle} \qquad \lambda : as' \vdash_{\mathsf{seq}_3} {}^{k}\mathcal{A}^{\langle 1,0 \rangle}}{\lambda : \mathsf{act}(d, as_a)::as' \vdash_{\mathsf{seq}_3} {}^{k}\mathcal{A}_d^{\langle 1,1 \rangle} + ({}^{k}\mathcal{A}_s^{\langle 1,0 \rangle} \| {}^{k}\mathcal{A}^{\langle 1,0 \rangle})}$$

$$\frac{}{\lambda : \emptyset \vdash_{\mathsf{seq}_3} {}^{k}\mathcal{F}^{\langle 2,0 \rangle}}$$

The rule $\vdash_{\mathsf{seq}}$, on the top, initializes the FSM, by receiving the event e given in the rule argument and adding the special seekers ℓ to the first location of the FSM. The rule $\vdash_{\mathsf{seq}}$ waits for the event λ together with e and creates a kill location which is the target of the *recv*-transition related to λ. The special seekers k is added to this kill location. Then, $\vdash_{\mathsf{seq}}$ applies the rule $\vdash_{\mathsf{seq}_3}$ to as with λ in parameter. The linker ${}^{\langle \ell,1 \rangle}\mathcal{L}$ merges the first location with one target of the *and*-transition.

The rule $\vdash_{\mathsf{seq}_3}$ treats similarly to $\vdash_{\mathsf{seq}_1}$ the parsed items, but stores the connector k in addition. Notice that when sub-sequence of actions $\mathsf{act}(d, as_a)$ are parsed, the rule $\vdash_{\mathsf{seq}_3}$ is applied to the sub-sequence as_a. At the end, instead of terminating with an ender $\mathcal{F}^{\langle 1,0 \rangle}$, ${}^{k}\mathcal{F}^{\langle 2,0 \rangle}$ terminates the FSM by deleting the connectors 1 and k. The FSMs constructed by these rules are depicted in Figure 4.22.

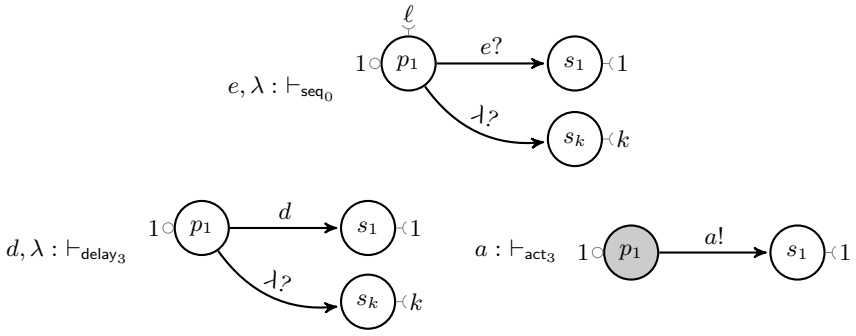

Figure 4.22: FSMs constructed by $\vdash_{\mathsf{seq}_3}$ and the new $\vdash_{\mathsf{seq}_0}$.

Example 4.2.26: We show the model $\mathcal{M}$ obtained by the application of the third construction rules to the AST depicted in Figure 4.15. The model $\mathcal{M}_{\mathsf{sys}}$ is depicted in Figure 4.23.

The linkers FSM, constructed with seekers ℓ and 1, merge a target of the *and*-transition with the first location, creating the required loop. The special connector k stores a unique location through the construction. It results in one kill location targeted by all the receive transitions of λ, for each FSM managing a sequence of actions triggered by an event. The kill location, can be used to manage a common sequence before terminating the FSMs. ◊

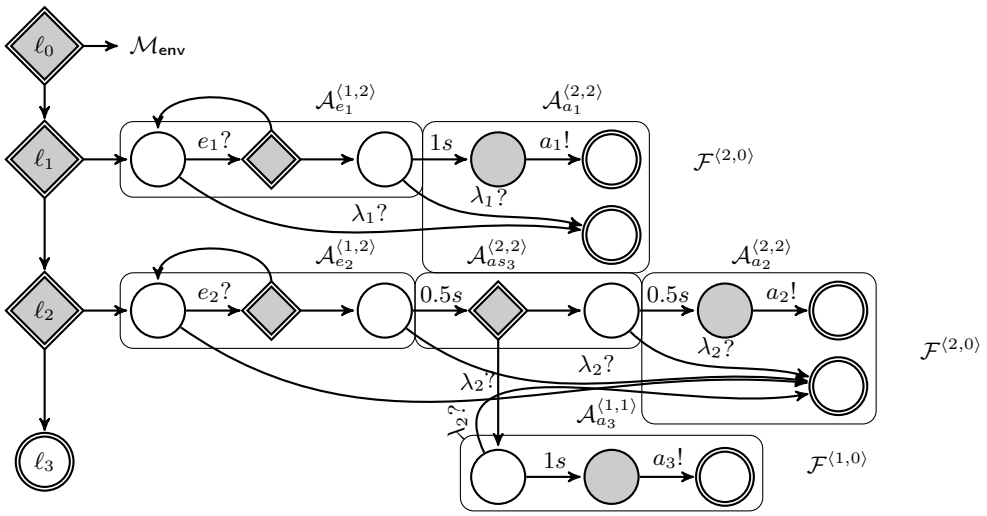

Figure 4.23: $\mathcal{M}_{\mathsf{sys}}$ resulted from the third example.

4.3 Input Generation Algorithms

The generation phase of our TMBT framework described in Section 4.1.3, must generate a suite of input traces $\mathcal{T}_{\mathsf{in}}$, defining the quality of the test.

In this section, we detail several algorithms in order to generate such a suite $\mathcal{T}_{\mathsf{in}}$. First, we focus on model-based algorithms in Section 4.3.1, and presents the existing algorithms used by our framework to construct a $\mathcal{T}_{\mathsf{in}}$ from a model $\mathcal{M}$. Then, we present, in Section 4.3.2, requirement-based algorithms. These algorithms are implemented for our musical application case, using a formal music representation to generate relevant suite of inputs. Thereafter, we introduce in Section 4.3.3 probabilistic models, de-

tailing another approach to generate relevant input suites with model-based algorithms.

We define the test quality as the coverage of the model $\mathcal{M}$ by its suite of test cases, for a test case $\langle \sigma_{\mathsf{in}}, \sigma_{\mathsf{ref}} \rangle$ as defined in Section 4.1.3. More precisely, for the set $\mathcal{T}_{\mathsf{ref}} = \{\mathcal{M}_{\mathsf{sys}}(\sigma_{\mathsf{in}}) \ \forall \sigma_{\mathsf{in}} \in \mathcal{T}_{\mathsf{in}}\}$ of reference traces, the test quality for a suite $\mathcal{T}_{\mathsf{in}}$ is the percentage of model items reached when computing the set $\mathcal{T}_{\mathsf{ref}}$ during $\mathcal{M}_{\mathsf{sys}}$ simulations.

Moreover, we define four kinds of algorithms to generate an input suite: *offline*, *online*, *model-based* and *requirement-based*. In the following of the section, we present some algorithms according to these four kinds and compare them using two characteristics: the *coverage* of a generated suite, and its *state explosion* when the size of the model $\mathcal{M}$ increases. We define *state explosion* as an impossibility to generate the suite $\mathcal{T}_{\mathsf{in}}$ because the memory or time required by the algorithm is too large.

4.3.1 Model-Based Algorithms

Given a model $\mathcal{M}$, model-based algorithms use $\mathcal{M}$ to generate a set of input traces $\mathcal{T}_{\mathsf{in}}$. This set is impacted by the possible input sequences permitted by the test environment $\mathcal{M}_{\mathsf{env}}$ in $\mathcal{M}$.

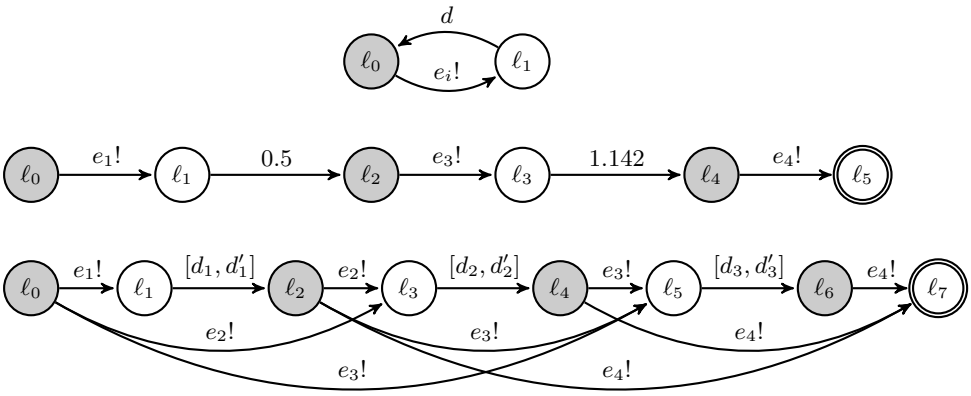

Figure 4.24: Three test environment examples: generic, singleton, and musically relevant with $n_{err} = 2$ and $d_i = d_i^e(1 - \kappa)$ and $d_i' = d_i^e(1 + \kappa)$.

For instance, the $\mathcal{M}_{\mathsf{env}}$ models, depicted in Figure 4.24, have three different test meanings:

- The first test environment model, on the top, is the most general and emits any event of any duration. Remark that this test environment is the generic one constructed in Section 4.2.3 with time management. However, the set of possible input sequences is huge and contains all the sequences (without simultaneous events) accepted by the model $\mathcal{M}_{\mathsf{sys}}$.

- The second test environment model is the strictest model and only allows one trace in $\mathcal{T}_{\mathsf{in}}$: $\sigma_{\mathsf{in}1} = \langle 0, e_1\rangle \cdot \langle e_3, 0.5\rangle \cdot \langle e_4, 1.642\rangle$. The algorithm should not have *state explosion* problems but the test quality is very low because the model items covered are strictly those of $\mathcal{M}_{\mathsf{sys}}(\sigma_{\mathsf{in}1})$.

- The last test environment model is relevant for musical cases and provides parameters n_{err} and κ to control the possible input sequences during generation. In the example, the events must be emitted in a chronologic order following the score, but permits some mistakes by missing until two consecutive events. Moreover, the event durations are chosen in an interval around the ideal duration, modeling the possible interpretations a musician can do when playing music.

Notice that $\mathcal{M}_{\mathsf{env}}$ models target a sub-set of real environment behaviors. This notion is useful to avoid obvious or impossible input sequences that are costly and useless in the tests. It is for example impossible to emit the event *quit* before *start*. Moreover, it is useful to reduce the test session in presence of *state explosion*.

Exhaustive generation

We present an *offline* and *model-based* algorithm provided in our TMBT framework thanks to an existing tool. The tool CoVer [18], of the Uppaal suite [44], is used in order to generate automatically an exhaustive suite of input traces $\mathcal{T}_{\mathsf{in}}$ from a network of Timed Automata and given some coverage criteria. To compute such a suite, the coverage criteria are translated into a finite state automaton $\mathcal{O}\mathsf{bs}$, called observer, monitoring the parallel simulation of the TA network. Each transition to a final state of $\mathcal{O}\mathsf{bs}$ is labelled by a predicate according to the coverage criteria. A predicate becomes true when the corresponding monitored item is reached in the simulation. Finally, a monitored item can be a transition, a location or a variable declaration/use in the TA network. Then, the model checker Uppaal is used by CoVer to generate the set of input traces $\mathcal{T}_{\mathsf{in}}$ resulting from an execution of the synchronized product of $\mathcal{M}'_{\mathsf{env}}$ and $\mathcal{M}'_{\mathsf{sys}}$ with $\mathcal{O}\mathsf{bs}$. The execution is performed

until the final states of $\mathcal{O}\mathsf{bs}$ are reached. $\mathcal{M}'_{\mathsf{env}}$ and $\mathcal{M}'_{\mathsf{sys}}$ are the translated Uppaal's format TAs from the IRTMs $\mathcal{M}_{\mathsf{env}}$ and $\mathcal{M}_{\mathsf{sys}}$, this translation is done under-restrictions and following the procedure described Section 3.2.

For loop-free IRTMs $\mathcal{M}_{\mathsf{env}}$ and $\mathcal{M}_{\mathsf{sys}}$, with an observer checking that all transitions of $\mathcal{M}'_{\mathsf{env}}$ and $\mathcal{M}'_{\mathsf{sys}}$ are fired, CoVer will return a suite of test cases complete for non-conformance: if there exists an input trace $\sigma_{\mathsf{in}} \in \mathcal{M}_{\mathsf{env}}$ such that $\sigma_{\mathsf{moni}} = \mathsf{IUT}(\sigma_{\mathsf{in}})$ and $\sigma_{\mathsf{ref}} = \mathcal{M}_{\mathsf{sys}}(\sigma_{\mathsf{in}})$ differ, then the suite of test cases will contain such an input trace.

In practice, we avoid state explosion with appropriate restrictions on $\mathcal{M}_{\mathsf{env}}$, using the parameters n_{err} and κ presented in Section 4.2.3 for the construction of $\mathcal{M}_{\mathsf{env}}$.

The main limitations of this offline approach are that:

(a) it does not scale well for testing real requirements,

(b) for our application case, the input traces are not musically relevant because CoVer strictly follows the model constraints and

(c) the IRTM must follow the translation restrictions.

However, this approach is well suited for debugging and regression tests, using small ad-hoc requirements.

Online generation

The second model-based algorithm is *online* and implemented in the VM. Online algorithms simulate the model, meanwhile the IUT is stimulated. It is a *model-based* and real-time testing approach which performs a loop of four actions:

- verify action if received from the IUT,
- send an event to the IUT,
- wait for a duration to advance some time,
- or validate the test by terminating it.

In practice, a set $\mathcal{Z}$ of possible next states in the model is computed and updated after each action. Hence, an online test case returns fail if $\mathcal{Z}$ becomes empty.

Remark that Tron [44], the existing Uppaal's tool for *online* testing, could not be used in our application. The reason comes from the restriction R_1 of the translation IRTM into TA in Section 3.2. Indeed, this transition requires that the Model Time Unit (mtu) of a TA network follows a tempo curve Φ. However, Tron can manage several clock rates only when they are defined as a constant factor of the wall clock. Hence, this constraint does not comply with the notion of an evolving tempo which is crucial in our case.

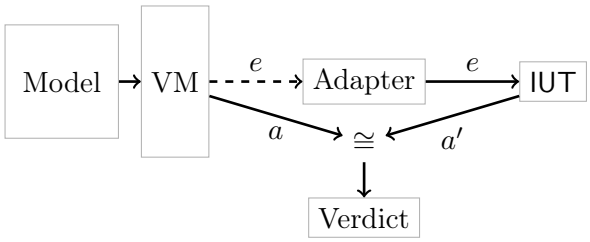

Figure 4.25: Online timed testing workflow

Figure 4.25 depicts our *online* timed testing approach. Instead of computing the set $\mathcal{Z}$, we use the determinism of IRTMs in its action emissions to send to a comparator ($\cong$) the reference and monitored output symbols, timestamped by the comparator when received. Indeed, only two possible non-deterministic cases can be seen in IRTMs: a *wait*-transition of the form $\ell \xrightarrow{[d(1-\kappa),d(1+\kappa)]} \ell'$ for some $0 \leq \kappa < 1$, and, branches with several *emit*-transitions. This non-determinism allows the online algorithm to choose: an appropriate value of duration to wait, or a symbol to emit, when the VM must provide an input (for resolving the cases (2), (3) and (4) of online algorithm actions). These choices correspond to the "on-the-fly" generation of an input trace σ_{in}.

Our online algorithm implemented in our TMBT framework contains four cases:

- One or more timestamped actions, $\langle a'_i, t'_i \rangle$, are received from the IUT. The corresponding model actions, $\langle a_i, t_i \rangle$, are searched and the equation $|t_i - t'_i| \leq \epsilon$ is tested. The inverse is performed if $\langle a_i, t_i \rangle$ is received, implying two lists of actions, say σ_o and σ'_o, stored in the comparator. The actions are retrieved from the lists if their related actions are found.

- Assume that the VM is in state $s = \langle t, n, \Gamma, pc, \theta \rangle$ and that an (emit) move can be executed. More precisely, assume that the running location $\Gamma[pc] = \langle \ell, \gamma, \beta \rangle$, where ℓ is a location of $\mathcal{M}_{\mathsf{env}}$, and that the branch at ℓ contains several *emit*-transitions with the symbols $e_1, \ldots, e_n \in Evt$. Then, the VM chooses randomly one of $e_1, \ldots, e_n$ and executes the (emit) move with it.

- Assume that a (delay) move can be applied to the state $s = \langle t, n, \Gamma, pc, \theta \rangle$ and the running location is $\Gamma[pc] = \langle \ell, \gamma, \beta \rangle$. The VM will choose

randomly a duration δ following the conditions of the (delay) move. More precisely, we compute the bounds δ_{min} and δ_{max} using the *wait*-transition $\ell \xrightarrow{[d,d']} \ell'$ as follows: $\delta = \Phi(d) - \gamma$ and $\delta' = \Phi(d') - \gamma$. Then, the VM chooses randomly δ in $[\delta, \delta']$ satisfying the conditions (2) and (3) of the (delay) move. Notice that if σ_o or σ'_o is not empty at this point, fail is returned from the test and the IUT does not conform the model. Indeed, we have reached the end of a logical instant and we found missed or unexpected outputs.

- The test case is terminated and pass is returned if $\sigma_o \cup \sigma'_o = \emptyset$.

The points (2), (3) and (4) are performed if no action is pending in (1). At the end, the missed (resp. unexpected) symbols are in the lists σ_o (resp. σ'_o).

In conclusion, the ad-hoc *online* method permits to test the IUT with IRTMs using the *online* approach, impossible with existing tools. It allows non-determinism on the model, dealing with state explosion by simulating in real-time the model and avoiding the storage of sets of traces (σ_{in}, σ_{ref} and σ_{moni} stored in the *offline* approach). Moreover, online testing is fast and support real cases. However, we still need to improve the coverage of the test cases related to our *online* generation algorithm. Therefore, it is not clear for now how to compute relevant input traces during *online* generation.

4.3.2 Requirement-Based Algorithms

In order to compute $\mathcal{T}_{\mathsf{in}}$, generation algorithms can focus on a set of most relevant input traces rather than to compute an exhaustive suite on a model $\mathcal{M}$. However, the relevance of input traces due to the requirements context can be lost or reduced when one abstracts the timed behaviors requirement into $\mathcal{M}$. Requirement-based algorithms generate a relevant $\mathcal{T}_{\mathsf{in}}$ from the requirements in order to use another, and more relevant, representations than the model $\mathcal{M}$.

This section presents a musical solution to generate relevant input traces via an *offline* algorithm. In the following of the section, we recall the musical sense of an input trace σ_{in} and present a relevant musical representation for such traces. Then, we use this representation during generation as a guide for our requirement-based algorithm.

A musical context

In a musical point of view, the document containing high-level requirements, is called a *score*. We define the *ideal trace* as the expected sequence of input events written in the score. It is the sequence of events that must be played by musicians.
For a sequence: $\nu^{ideal}_{t_{\mathsf{in}}} = e_1! \cdot d_{e_1} \cdot e_2! \cdot d_{e_2} \cdot \ldots \cdot d_{e_{n-1}} \cdot e_n!$, we define the related ideal trace $\sigma^{ideal}_{\mathsf{in}}$:

$$\langle e_1, 0\rangle \cdot \langle e_2, d_{e_1}\rangle \cdot \ldots \cdot \langle e_n, \Sigma_{n-1}^{i=1} d_{e_i}\rangle$$

We recall that usually, ideal traces are in an abstract time, called beat. During performance, the ideal trace will be "interpreted" by musicians given a concrete input trace in physical time more or less far from the ideal one. This notion of performance, presented concretely in Section 2.1, is formally defined by timing functions.

Timing Function. A theory has been proposed to define a performance using *Timing Functions* (TIFs). Known as Time-maps [59], Time-warps [39] or Time-deformations [5], TIFs are monotonically non-decreasing functions mapping score durations (in beats) in performed durations (in seconds). A TIF is a pair of *time-warping* functions $\langle \mathsf{f}^{\times}, \mathsf{f}^{+}\rangle$, the former defines a tempo-curve function, a tempo variations through a performance. The latter is a time-shift function and defines local event's timestamp alterations such as a *swing movement*. The first intuition is to consider three aspects of a music performance: the temporal structure (of the score rhythms), the global musician tempo and the local timing variations. Although *tempo* and *time-shift* are mathematically related, they are very different musical notions. Moreover, a composition operator is defined in order to sequentially compose several TIFs to build relevant musical performances.

The language Nyquist [40] comes from the *time-warping* formal notion, and implements such operators called *stretch* and *shift*. The first deals with tempo variations and the second expresses operations such as delay, rest or pause. As an example, the operation *(at 10 (stretch 2 (score)))* in Nyquist performs the score with a pace divided by two and event dates shifted by 10 seconds. However, in these functions the temporal structure is lost on the opposite of TIFs which track the score structure.

In order to use TIFs to represent performances, we detail the application of such functions to a relative time trace as defined Section 3.1. Given a relative timed traces in performance time σ_{in} composed of triples $\langle e, t, p\rangle \in$

$Evt \times t^{\sigma_{\mathsf{in}}} \times \mathbb{R}_*^+$, the application of a TIF $\langle \mathsf{f}^\times, \mathsf{f}^+ \rangle$ returns another relative timed trace σ'_{in} such that:

$$\langle \alpha_i, \mathsf{f}^+(t_i), \mathsf{f}^\times(p_i) \rangle, \ \forall \ \langle \alpha_i, t_i, p_i \rangle \in \sigma_{\mathsf{in}}$$

Intuitively, the time-shift function f^+ modifies the event musical timestamps whereas the tempo curve $\mathsf{f}^\times$ alters the tempo values of a musical trace.

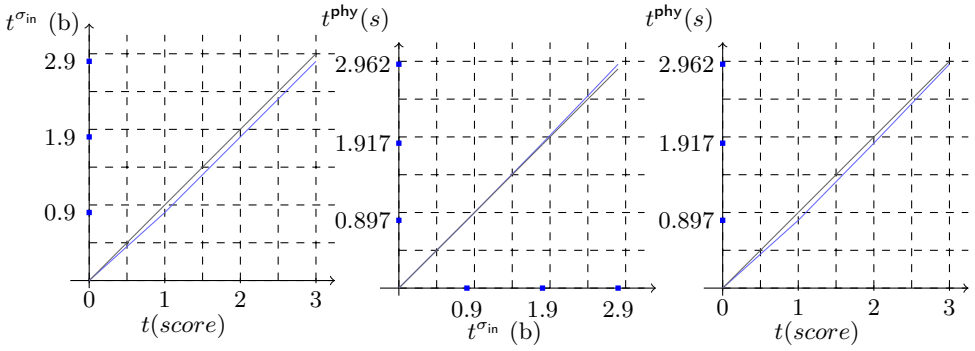

Figure 4.26: Example of a TIF, from left to right: the time-shift function f^+, the tempo curve $\mathsf{f}^\times$ and the corresponding TIF. Gray lines depict ideal values.

Example 4.3.27: Given a mixed score specifying a sequence of four quarters (each note lasts one beat) played with 60_{bpm}, the ideal trace is $\langle e_1, 0, 60 \rangle \cdot \langle e_2, 1, 60 \rangle \cdot \langle e_3, 2, 60 \rangle \cdot \langle e_4, 3, 60 \rangle$. From this trace, we want to describe a performance inducing: a *swing*, shifting the event timestamps to 0.1 beat earlier, and, a decelerating tempo from 60.2 to 56_{bpm}. The performance is formally defined by:

- a time-shift of 10%, and
- a tempo curve decreasing at each event detection.

We recall that a relative trace defines piecewise constant tempo curves. Hence, these functions are applied to each event, which is each item of an input trace.

These two functions are depicted in Figure 4.26 and detail their application to our ideal trace: f^+, on the left, shifts the ideal timestamps, and $\mathsf{f}^\times$, on the middle, alters the tempi and translates performance time into

physical time. The TIF is depicted on the right and applies the previous two functions. The result is the relative input trace $\sigma_{\mathsf{in}1}$:

$$\langle e_1, 0, 60.2\rangle \cdot \langle e_2, 0.9, 58.8\rangle \cdot \langle e_3, 1.9, 57.4\rangle \cdot \langle e_4, 2.9, 56\rangle$$

In order to compute the corresponding physical timestamps, we apply the translation function Φ to the relative event durations with the current tempo: $d_i^{\mathsf{phy}} = (d_i^{\sigma_{\mathsf{in}}} * 60)/p_i$, with $d_i^{\sigma_{\mathsf{in}}} = t_i - t_{i-1}$.
Finally, we obtain the physical sequence: $\nu_1 = e_1! \cdot 0.897 \cdot e_2! \cdot 1.020 \cdot e_3! \cdot 1.045 \cdot e_4! \cdot 1.071$ resulting to the physical input trace $\sigma_{\mathsf{in}1}{}'$:

$$\langle e_1, 0\rangle \cdot \langle e_2, 0.897\rangle \cdot \langle e_3, 1.917\rangle \cdot \langle e_4, 2.962\rangle$$

$\Diamond$

The TIF representation is a good solution to produce relevant performances, we have so a formal representation and can compose them to construct other performances. Moreover, it is well adapted to our test traces representation of performances because each function is applied to a specific field of a relative input trace.

Translation. Notice that a physical trace can be the translation of an infinity of relative time traces. We saw that the translation of $\sigma_{\mathsf{in}1}$ is $\sigma_{\mathsf{in}1}{}'$. However, we can have the same physical trace from other relative traces. For example, one specifying only a tempo change, and, the other only time-shifts:

$$\sigma_{\mathsf{in}2} = \langle e_1, 0, 66.889\rangle \cdot \langle e_2, 1, 58.823\rangle \cdot \langle e_3, 2, 57.416\rangle \cdot \langle e_4, 3, 56.022\rangle$$
$$\sigma_{\mathsf{in}3} = \langle e_1, 0, 60\rangle \cdot \langle e_2, 0.897, 60\rangle \cdot \langle e_3, 1.917, 60\rangle \cdot \langle e_4, 2.962, 60\rangle$$

We can easily compute these traces from the equations below because our ideal trace considers a tempo $p_i = 60_{\text{bpm}}$ and event durations of $d_i^{\sigma_{\mathsf{in}}} = 1$ beat.

$$d_i^{\sigma_{\mathsf{in}}} = (d_i^{\mathsf{phy}} * p_i)/60 = d_i^{\mathsf{phy}}$$

with $p_i = 60$.

$$p_i = (d_i^{\sigma_{\mathsf{in}}} * 60)/d_i^{\mathsf{phy}} = 60/d_i^{\mathsf{phy}}$$

with $d_i^{\sigma_{\mathsf{in}}} = 1$. The equations are computed from the function $\Phi^{\sigma_{\mathsf{in}}-1}$. However, with these traces, it is much harder to understand what was the musician behavior during performance than with the trace $\sigma_{\mathsf{in}1}$.

Interpretation. Given a relative input trace σ_{in} of triples $\langle e_i, t_i, p_i \rangle$, we can have two different interpretations:

- a musician plays the event e_i at t_i beat with a tempo of p_i, or
- the musician's event e_i is detected at t_i beat with a tempo of p_i.

The first interpretation is musical and relevant when, for example, we describe performances with TIFs. Whereas, the second is comprehensible for a model input, where the input trace is a detection of a continuous IMS module. However, it implies a different computation of physical values:

- in the first case, the equation to compute physical durations is: $d_i^{\mathsf{phy}} = (d_i^{\sigma_{\mathsf{in}}} * 60)/p_i)$,
- in the second it is: $d_i^{\mathsf{phy}} = (d_i^{\sigma_{\mathsf{in}}} * 60)/p_{i+1}$.

Indeed, in the second case the onset of e_{i+1} is detected at t_{i+1} with the tempo p_{i+1}. However, the detected tempo is in fact the e_i's.

We distinct these two input trace interpretations by calling *performances* the first traces and *recognition* traces the seconds. In this section, we dealt with *performances*.

Fuzzing algorithm

Given an ideal trace $\sigma_{\mathsf{in}}^{ideal}$ interpreted as a performance, the fuzzing algorithm consists of starting with $\sigma_{\mathsf{in}}^{ideal}$ associated to a mixed score and adding deformations of several kinds. For this purpose, we use TIF representations in order to create musically relevant performances.

The implemented fuzzing function takes in input an ideal trace and parameters for bounding the deviations on the time-shifts, the tempo values and the number of missing notes. It generates some random values within theses limits and applies them to return an input trace σ_{in}' as a mutation of the $\sigma_{\mathsf{in}}^{ideal}$.

An interesting open question in this context is the definition of TIFs for the generation of covering test suites following criteria similar to those of the model-based algorithms. The advantages of the requirement algorithm are their easy and fast execution. However, the algorithm cannot provide yet a good coverage by the input trace generated.

4.3.3 Stochastic Algorithms

Computation of relevant sets of input traces $\mathcal{T}_{\mathsf{in}}$ can be done with model-based algorithms. A solution consists in adding probabilities to the model for specifying the relevance of transitions.

This section presents model-based algorithms using probabilities in $\mathcal{M}$ to generate relevant input traces. First, an implemented algorithm is proposed following a brute-force approach. Then, a smarter algorithm is detailed to avoid systematic and useless simulations during the generation of $\mathcal{T}_{\mathsf{in}}$.

An extension of TA, the Probabilistic Timed Automata [65] (PTA), relates a state and an action to a probability distribution over the set of states. In short, a probability distribution over a set $\mathcal{X}$ is a function that assigns a probability in $[1, 0]$ to each element of $\mathcal{X}$, such that the sum of the probabilities of all elements is 1. Then, during simulation of probabilistic models, solving functions, called *adversaries*, are in charge of computing and choosing a path in a set of probabilistic edges. In [49], a probabilistic ioco conformance (pioco) is formally defined for performing probabilistic MBT, allowing formal test procedures for non-deterministic systems.

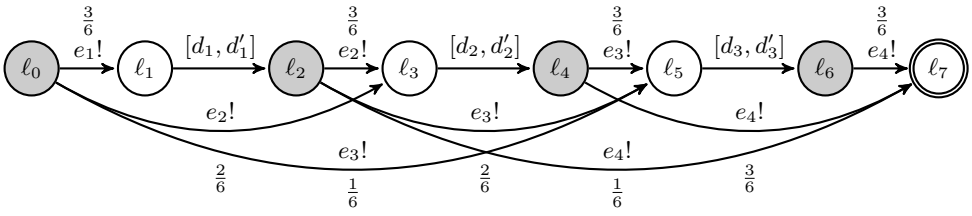

Figure 4.27: Probabilistic test environment example.

Probabilistic models can be used to specify the relevance of input sequences in test environments $\mathcal{M}_{\mathsf{env}}$. Indeed, the relevance of a trace can be computed during its generation in order to drop traces with a low probability. Figure 4.27 depicts an example of IRTM with probabilities. The environment model $\mathcal{M}_{\mathsf{env}}$ has probabilistic values on its *emit*-transitions. In this example, the fact of sending an event $i + n$ with $n > 1$ and missing at least one event has a lower probability than playing a performance without mistakes. This solution prevents from generating input traces with lots of missed events, which is musically irrelevant. Moreover, the choice of duration values can follow a normal function with a mean set to the ideal duration of events. Hence, more the generated value δ within these bounds is far from

the ideal value, more its probability is low. Combining these two adversaries, resolving the event and duration choices, improves the model-based algorithms coverage without managing complex functions.

Uppaal-SMC. SMC [43] is another extension of Uppaal to specify dynamics and stochastic behaviors of systems. This extension manages a multi-rate clock set, probabilistic transitions and uses the Statistical Model Checking (SMC), an alternative model checking technique to avoid an exhaustive exploration of the model state-space. Thanks to this tool, two algorithms based on probabilistic models were implemented to improve generation of relevant input traces.

1) Brute-force Algorithm

Given a probabilistic model $\mathcal{M}_{\mathsf{env}}$, specifying the relevance of its possible paths, via probabilities or/and intervals of durations. The first idea is to generate N input traces σ_{in} from $\mathcal{M}_{\mathsf{env}}$ and compute their frequency.

In order to gather several physical traces, we define a set of percentages of event duration $\mathcal{K} = \kappa_1, \ldots, \kappa_n$, such that $0 < \kappa_i < \kappa_{i+1} < 1$. The set $\mathcal{K}$ is a set of regions gathering a range of durations considered as having a same probability. The algorithm requires the ideal event durations of the score and proceeds as follows:

1. First, N stochastic simulations are performed with $\mathsf{Uppaal-SMC}$, to generate the set of input traces $\mathcal{T}_{\mathsf{in}}$ following the probabilities in $\mathcal{M}_{\mathsf{env}}$.
2. Then, we construct a tree $\mathcal{T}$ to gather and count the trace frequencies. $\mathcal{T}$ is a tree with weight nodes and alternating *event*- and *region*-edges, denoted $t \xrightarrow{e_i} t'$ and $t \xrightarrow{\kappa_i} t'$, with $e_i \in Evt$, $\kappa_i \in \mathcal{K}$ and t, t' two weighed tree nodes. We denote t_{++} the incrementation of the weight in the node t.
3. For each input trace $\sigma_{\mathsf{in}} \in \mathcal{T}_{\mathsf{in}}$, we start from the root of $\mathcal{T}$ and construct or traverse it, according to the input sequence ν_i related to σ_{in}.

We recall that an input sequence is a sequence of $e_i! \cdot d_i$, with $e_i!$ the event emitted and d_i its duration. The algorithm counts the frequency of an input sequence ν_i by handling the sequence of items in three cases:

1. $e_i{::}\nu'$, an event e_i is parsed, the transition $t \xrightarrow{e_i} t'$ is taken,
2. $d_i{::}\nu'$, a duration is parsed, we take the transition $t \xrightarrow{\kappa_i} t'$, where $d_i^e * \kappa_{i-1} < d_i < d_i^e * \kappa_{i+1}$ with d_i^e the ideal duration of e_i, or

3. the end of ν_i is reached, we execute t_{++}, the weight of the current node is incremented.

Finally, we obtain the frequency of each set of κ-similar traces pr_i with the equation:

$$pr_i = \frac{t_i}{|\mathcal{T}_{\mathsf{in}}|}$$

for each node in $t_i \in \mathcal{T}$ (excepted the root).

Then, given a threshold γ, the traces with less than γ frequency are retrieved from $\mathcal{T}_{\mathsf{in}}$. The algorithm has a complexity of $\mathcal{O}(2 * N)$ in time and $\mathcal{O}(N)$ in space, with $N = \Sigma_{i=0}^{n} |\sigma_i| \in \mathcal{T}_{\mathsf{in}}$, the size of all the input traces in the set $\mathcal{T}_{\mathsf{in}}$. Indeed, it implies N simulations and traversals of each trace in $\mathcal{T}_{\mathsf{in}}$.

2) Back-Counting Algorithm

The second algorithm prevents from computing an input trace and discards it at the end (because of a too low probability). Given a threshold γ, the second algorithm generates only the set $\mathcal{T}_{\mathsf{in}}$ such that, $\forall \sigma_{\mathsf{in}} \in \mathcal{T}_{\mathsf{in}},\ pr(\sigma_{\mathsf{in}}) \geq \gamma$, with $pr(\sigma_{\mathsf{in}})$ the probability to have the input trace σ_{in}. The idea is to associate, in the last model location of $\mathcal{M}_{\mathsf{env}}$, the value γ, and computes the minimum $pr(\sigma_{\mathsf{in}})$ required to fulfill $pr(\sigma_{\mathsf{in}}) \geq \gamma$ at each model location.

The algorithm rewrites the model $\mathcal{M}_{\mathsf{env}}$, from its last location to its initial location ℓ_0, adding:

- a condition into guards and
- an assignment into updates

to each model transition.

More precisely, let the variable pr be the probability associated to a model location, and denoting an abstract model transition with $\langle \ell', pr' \rangle \rightarrow \langle \ell, pr \rangle$ where ℓ', ℓ are model locations and pr', pr are their associated probability variable before and after the transition. The variables pr are initialized to $\top$, the greatest value, for all the locations.

Moreover, a variable p stores the current probability of σ_{in}, during its generation and is initialized to 1 in location ℓ_0. Following the breadth first step method, the algorithm executes the following rewriting rules:

$$\langle \ell', pr' \rangle \xrightarrow{\beta} \langle \ell, pr \rangle \rightsquigarrow \langle \ell', pr'' \rangle \xrightarrow{\beta,\ p \geq pr',\ p = p * \beta} \langle \ell, pr \rangle$$

with $pr' = \frac{pr}{\beta}$, $pr'' = \min(pr'', pr')$ and $\beta \in [0, 1]$.

$$\langle \ell', pr' \rangle \xrightarrow{[d_{min}, d_{max}]} \langle \ell, pr \rangle \rightsquigarrow \langle \ell', pr'' \rangle \xrightarrow{[d_{min}, d_{max}],\ p \geq pr',\ p = p * D} \langle \ell, pr \rangle$$

with $pr' = \frac{pr}{D}$, $pr'' = \min(pr'', pr')$ and $D = pr(\delta) \in [0, 1]$ is the probability of the duration chosen when the transition is fired.

The first rule traverses *emit*-transitions with a probability of β. The rule updates the variable pr'' according to β and pr. Then the rule prohibits to fire the transition if p is too low thanks to the guards on p. The second rule traverses *wait*-transitions and performs the same modifications but with the probability of δ, the duration generated by the adversary.

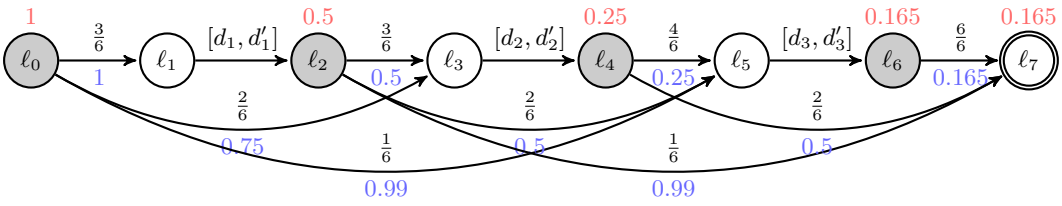

Figure 4.28: Probabilistic test environment after rewriting, omitting input event emissions.

For instance, the model $\mathcal{M}_{\mathsf{env}}$ depicted in Figure 4.27 is rewritten with $\gamma = 0.165$. The result is depicted in Figure 4.28. Blue values, under transitions, are the upper bounds of p in the guard transitions ($p \geq pr'$) and red values, over locations, are the minimum values of pr'' during the rewriting procedure.

In the example, the three transitions can be taken in location ℓ_0, hence, e_1 and e_2 can be missed. In location ℓ_2, if e_1 was played, e_2 can be missed, because p should be $0.5 = 1 * \frac{1}{2}$ and the guard $0.5 \geq 0.5$ holds in order to go to ℓ_5. However, the third transition to ℓ_7 cannot be taken because the guard requires a probability more than 0.99 and we have $p = 0.5$. No missed event can be done in location ℓ_4 for the same reasons.

This second algorithm is more promising than the first one because just the model is parsed rather than $\mathcal{T}_{\mathsf{in}}$. Hence, the algorithm complexity is lower and near $\mathcal{O}(t * N)$, with t the number of $\mathcal{M}_{\mathsf{env}}$ transitions and N the number of the generated input traces in $\mathcal{T}_{\mathsf{in}}$.

Unfortunately, this kind of model cannot be simulated with SMC because probabilities and variables are managed separately in the current version.

Moreover, this method is not mature enough to choose a good function of delay probability, needed to map duration in probability with a mean on the ideal value. This investigation has been performed during an internship in the team Formal Method and Tools (FMT[1]) of Twente University, Netherlands.

Summary

Based on IRTMs, we presented in this chapter an automatic and timed MBT framework. This framework allows to assess the timed conformance of real-time systems to their model, providing several parameters to configure test sessions.

We detailed the different steps composing our TMBT framework applied to abstract real-time systems. We presented what we mean by high-level requirements and how it is used in the construction step. Thereafter, the usual MBT steps were detailed and we explained our algorithm to compare two output traces containing logically timestamped actions. Next, we presented our construction rules and their compositional construction of IRTMs, giving examples of $\mathcal{M}_{\mathsf{env}}$ and $\mathcal{M}_{\mathsf{sys}}$ model constructions. Moreover, we dealt with algorithms to generate a suite of relevant input traces and presented existing tools or implemented approaches to compute musically relevant traces. Finally, we introduced stochastic values in models to construct relevant traces during simulations.

Through these MBT steps, the framework parameters allow to: delimit test sessions, chose the generation algorithm, and, fix the tolerance in comparison. Our TMBT framework was presented in a general manner. In the following of the manuscript, we detail more concrete steps by applying the testing procedure to a case study.

[1] `http://fmt.cs.utwente.nl`

Chapter 5

Case Study: Application to the Interactive Music System Antescofo

Our TMBT framework is based on high-level requirements to assess the timed conformance of real-time systems. Score-based IMS are real-time systems and must follow the temporal requirements of their mixed-score given in input before a performance. This chapter aims at assessing, with our TMBT framework described in Chapter 4, a state-of-the art score-based IMS conformance *wrt.* a given mixed-score.

More precisely, given our TMBT framework and a mixed-score, we want to assess the IMS Antescofo conformance to the mixed-score temporal requirements.

Antescofo stands for *anticipatory score following* and is an electronic musician which plays music according to the timed scenario specified by the mixed score with respect to a human live performance. Incorporate testing in the development of such a system is a challenge for several reasons:

- As depicted in Section 2.1, in general and despite strong real-time requirements, testing methods are not well developed for such music systems yet;
- Antescofo is a real-time system and is involved in live music performance. Its time reliability is a critical point to prevent from misbehaviors during performances and concerts.

- Antescofo is in a constant evolution, because it is used in live performances by famous composers, pushing forward the system to meet new constraints and requirements.

As a consequence, the need of relevant and trustable testing framework, especially for regression testing, appeared soon as essential for the well development of Antescofo.

This chapter first introduces the score-based IMS Antescofo in Section 5.1. Then, in Section 5.2, we exhibit our rules for model constructions based on a fragment of the Dedicated Specific Language (DSL) of Antescofo. This section presents the concrete implementation of our TMBT framework, such as the adapters to stimulate the IMS, the models constructed from an Antescofo mixed-score toy example and the verdict outputs. Finally, in order to evaluate the framework, we apply our testing procedure to Antescofo with several parameters for a benchmark containing both real mixed-scores and small mixed-scores in order to test the system. We report the results of our experiments in Section 5.3.

5.1 Antescofo

Along our work, our TMBT framework was applied to a state-of-the-art Interactive Music System. This IMS, Antescofo [46], is a score-based IMS taking a timed requirement, called a *mixed-score*, in input for performing a musical piece jointly with human musicians. More precisely, during performance Antescofo listens to a musician playing the instrumental part of the mixed score. Meanwhile, the system detects the musician's position in the given mixed score and the musician's instantaneous pace (its tempo, in beat per minute). Moreover, Antescofo plays in response the electronic part of the mixed score. Finally, this can be seen as an electronic accompaniment.

This section describes the Implementation Under Test (IUT) used in this case study. The specific architecture of Antescofo is first detailed in Section 5.1.1. Then, Section 5.1.2 presents the IMS Domain Specific Language (DSL) used to specify expressive mixed-score.

5.1.1 Architecture

Antescofo was created in 2007, it first proposed to connect a listening machine with a domain specific language to express mixed-score, at composition time, and to implement it, at performance time. It was developed then by the

INRIA team MuTant in the RepMus team of the Institute of Research and Coordination Acoustic/Music (IRCAM).

In this section, we just briefly overview Antescofo and its musical background. For more details, the reader can refer to [46], the Antescofo documentation[1], or follow the team's website link[2].

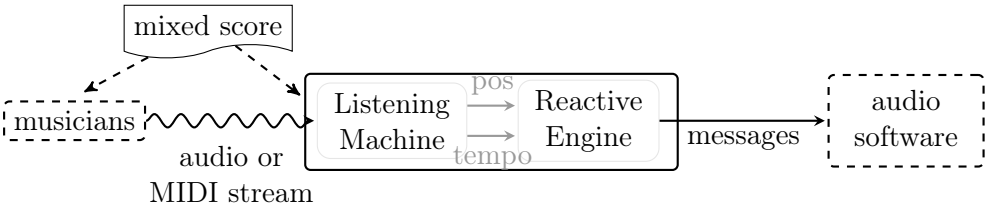

Figure 5.1: Architecture of Antescofo

The architecture of Antescofo is roughly depicted in Figure 5.1 and consists in two main modules: a Listening Machine (LM) and a Reactive Engine (RE). First, the LM is in charge of decoding the audio or midi input stream incoming from musicians and must infer in real-time:

(i) the musician's position in the given mixed score, and
(ii) the estimated musician's pace (or *tempo* in beats per minute).

These values are sent to the RE which schedules the electronic actions to be played, as required in a mixed score. For Antescofo, the actions are messages emitted on time to an audio environment. Hence, using external audio applications, sound emission/generation/transformation can be done. Moreover, Antescofo can send messages using standard network communication protocols, hence, any systems accepting UDP messages can be targeted by Antescofo messages.

Antescofo workflow. Usually, Antescofo workflow consists first in writing a mixed-score in the Antescofo's DSL and giving it to the system as argument. From here, Antescofo's LM is active and waits for audio signal or midi input stream to detect whether one of the first specified events is played. The listening machine [35] implements a semi-Markov model to manage the position estimation according to the advancement of time and events' observations [37]. This probabilistic model is combined with a tempo extraction

[1] `http://support.ircam.fr/docs/Antescofo/manuals/`
[2] `http://repmus.ircam.fr/antescofo`

algorithm, based on Large's work [66], to smoothy estimate the tempo variations. It gives the software a fair and stable recognition module to follow and anticipate musicians during performance. It also eases the RE reactions, implementing in real-time electronic parts from a reliable LM sequence of outputs.

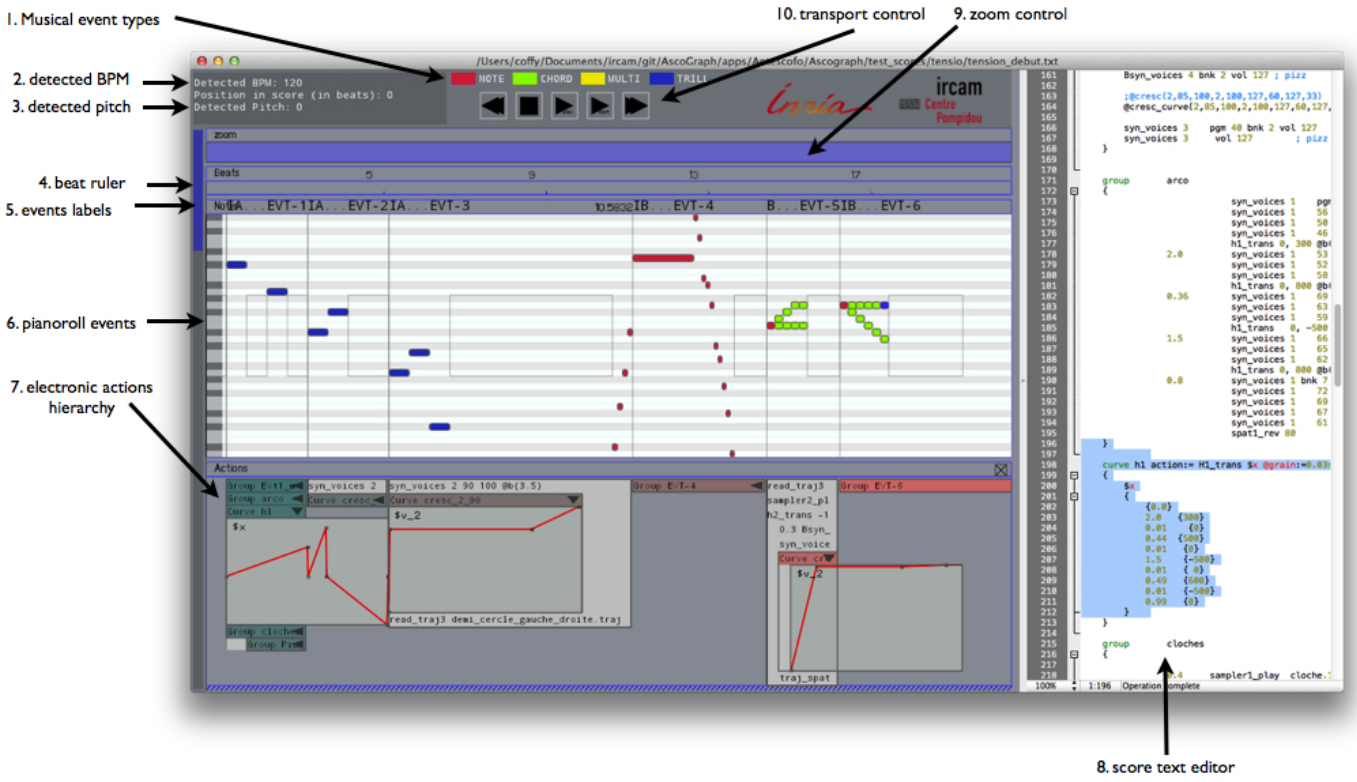

Figure 5.2: Ascograph: improving Antescofo mixed score composition and visualization.

The mixed-score composition and visualization are improved by a graphical extension of Antescofo called Ascograph [33]: Ascograph deals with a

visual representation of the multi-timed mixed-score and its dynamic variations during performance. Recently, an extension work [30] allows several views on Ascograph to focus on the representation of these dynamic variations which are untimed and do not follow a standard score timeline. An example can be found in Figure 5.2 showing on a score from *Philippe Manoury*. Briefly, the representation is composed of:

- a standard piano roll, on the middle, depicting the loaded mixed-score events. The piano roll representation displays events as rectangles where y-positions correspond to pitches, x-positions correspond to score positions and lengths correspond to durations. During performance, a vertical line informs about the current position of the musician for Antescofo (the last detected event),
- the Antescofo's DSL mixed-score editor, on the right. Parts of mixed-score can be dynamically loaded, modified and sent to Antescofo, permitting untimed actions launching, or Antescofo command emissions during a performance, achieving live-coding [90],
- the electronic actions to trigger are depicted on the bottom. It is the visual representation of the electronic parts, providing code snippets to create Antescofo actions.

To enable an easy and quick manipulation of the software and its sound output, Antescofo was first developed as a MAX-MSP [78] or PureData [79] object to be embedded in a patch. Patches are visual programs making accessible the specification of complex control and audio-processing graphs to not-expert users. Patches also simplifies drastically the connection with other softwares that are used in sophisticated set-up. The real-time system's behavior can easily be monitored thanks to the visual representation of these languages. As an example, the *reference* patch of Antescofo in MAX-MSP is depicted in Figure 5.3. The reference patch is built as an example for general uses of the Antescofo object in a patch and serves also as a documentation. This view explains the usual utilization of Antescofo in MAX-MSP by loading a score (top-left panel), starting Antescofo (the square below this panel) and launching the recognition on a pre-recorded performance or on a live audio stream (right side panel). During performance, panels on the bottom of the figure (connected to the outputs from the patch called Antescofo~), depict Antescofo current internal variables to track and monitor the system behaviors.

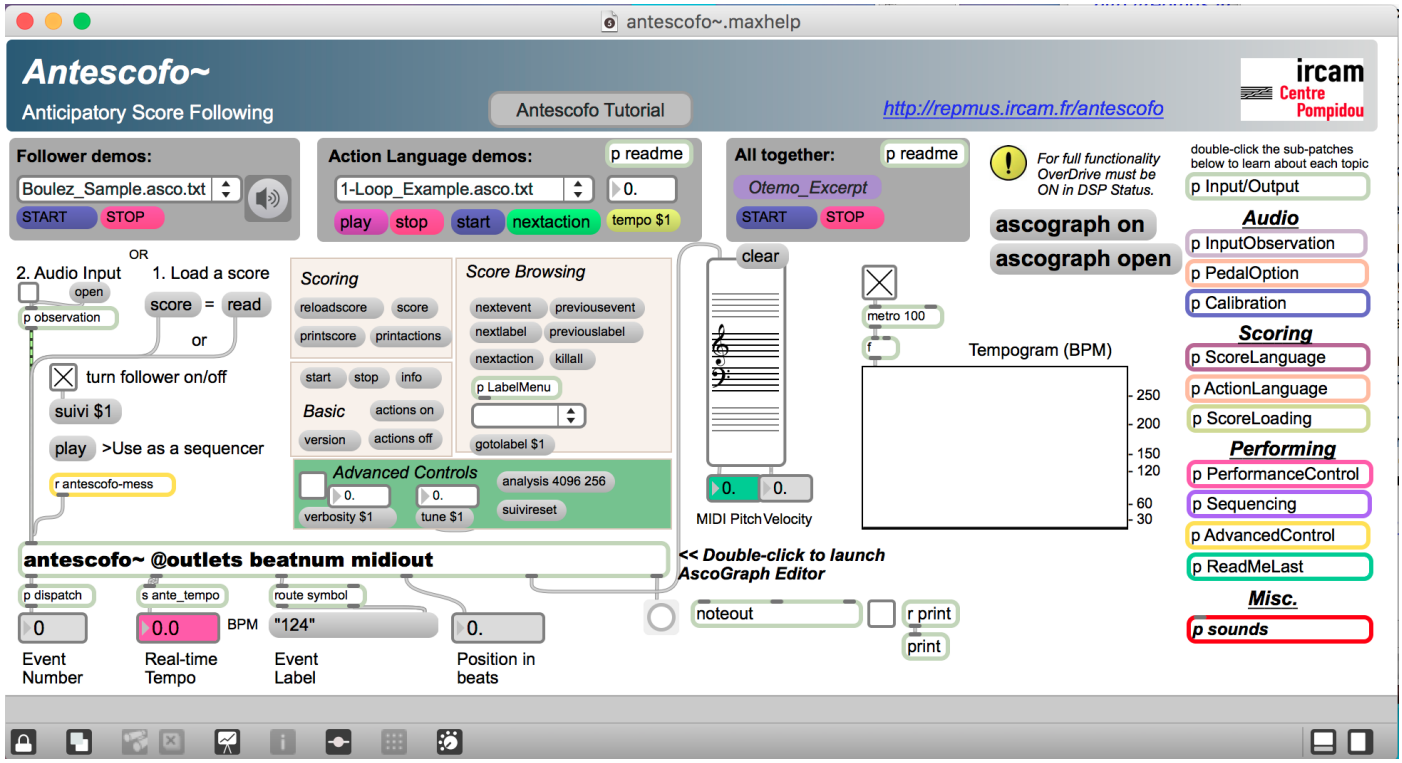

Figure 5.3: Antescofo reference MAX-MSP patch

Reactive Engine

The RE is the deterministic part of Antescofo and must implement in real-time the mixed-score given in input before the performance. By deterministic, we mean that for a same recognition trace detected by the LM, the timed reactions of the RE should be identical. However, the timed inputs of a musician performer is non-deterministic and interpretations, added to the ideal performance specified in the mixed score, differ for each performance. We mean by musician interpretations, the timing value variations/fluctuations of a performance from the ideal values.

The RE is an interpreter of the DSL mixed-score of Antescofo. The mixed-score, viewed as a program, is evaluated in real-time and is a sequence of timed computations that are synchronized with the events specified in the mixed-score and the tempo of the human performer. The challenge of the RE is to deterministically react from any input event sequences detected by

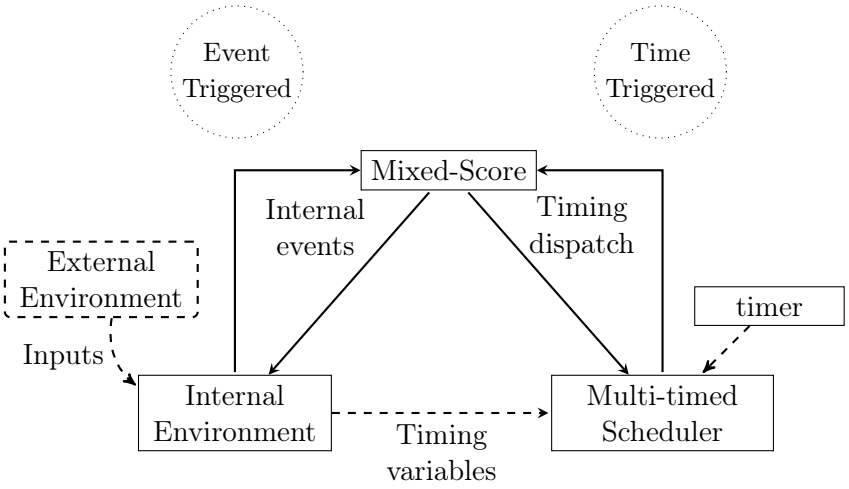

Figure 5.4: Architecture of the Antescofo reactive engine

the LM. To meet this challenge, the RE contains several components.

During performance and as depicted in Figure 5.4, the RE can receive input events, from the external environment, such as LM notifications of the musician position and tempo, or user's external commands. These inputs are variable updates managed by the internal environment. In the IMS, any variables are stored with their timed history if needed for latter access. Timing variables, which are clocks for specifying delays and durations in the mixed-score, are inserted in a priority queue gathering all the pending delay values. The time scheduler, according to the variables' priority, computes the minimum values in the queue in order to wait for the minimum duration until the next computation. Computations, launched by an event update (event-triggered) or a timing variable expiration (time-triggered), provoke other variable updates or timing dispatches in respect to the musician performance and the interpretation of the mixed score. Finally, an Antescofo run is a sequence of logical instants, defined by three possible inputs:

- an event detection (by the LM),
- a variable update (or a command) and
- a delay expiration.

Logical instants contain a finite sequence of instructions and are logically timestamped with the date of its beginning in physical time and an index number in order to differ logical instants with a same timestamp.

5.1.2 Antescofo Domain Specific Language

Mixed music pieces interleave musician and electronic timed behaviors. In this aim, a mixed-score specifies two parts:

- input events, expected from musicians, with their ideal durations, and
- output actions, the sequences of audio computations in reaction to the input events.

Attributes, denoted with @-words in the DSL, are used to add information to events and actions. In particular, we consider three kinds of attribute: *@name*, defines the label of an item; *@sync*, assigns the synchronization strategy to an action; and *@err*, assigns the error management strategy to an action.

By default, actions immediately following an event are triggered synchronously with the occurrence of this event. The other actions are triggered by the performance of their previous action after the expiration of an optional delay. This basic behavior, is modulated by the synchronization strategies defined with the attribute *@sync* which can have two values:

- *@loose*, following the default case, actions are triggered by the performance of their previous actions. The synchronization with musicians is managed only via delay fluctuations according to the detected tempo.
- *@tight*, instead of starting a delay after a previous action, the delay must be translated into an equivalent delay elapsing from the nearest event in the past. Here, equivalent means that the new delay is adjusted to achieve the same execution date if the event occurs at its specified date.

To introduce the DSL, we first present a simple example to briefly explain the syntax and ideas behind this language. Antescofo's DSL is a sequential textual language representing a mixed-score which contains temporal expressions.

```
1  NOTE C4 1 @name e1
2    a1 @name start
3    group @tight @name g1
4    {
5      1 a2 @name end
6    }
```

```
7
8  CHORD (D4 F4) 1 @name e2
```

Example 5.1.28: The code above is an Antescofo mixed-score containing two events (the note e_1 (line 1) and the chord e_2 (line 8)) and three actions (two *atomics* actions a_1 (line 2) and a_2 (line 5) and one compound action, the *group* g_1 (line 3)).

The events are defined with a type (note or chord), followed by one or a list of pitches (a C4 for the note e_1 and a D4 together with a F4 for the chord e_2). The next real specifies the event duration, being of 1 beat in the example. Finally, the attribute @name sets a precise label for these events.

The actions' specification begins with an optional value for their delay, it is 0 for a_1 and g_1 (0 is the value when the delay is not specified), and 1 beat for a_2. Then, the type of action is detailed, by default atomic actions (a_1 and a_2), and group for g_1. Similarly to events, the attribute @name sets the actions' labels. An attribute @tight is seen for g_1, specifying its synchronization strategy.

According to the synchronization attributes, the action a_1 is related to the event e_1 and because no delay is specified for a_1, the action is performed at the detection of e_1 (resulting in a message emission). Then, the next action (the group g_1) has a synchronization @tight and should be triggered according to the occurrence of e_1. Similarly to a_1, g_1 is performed at the detection of e_1 because no delay is specified (resulting in the management of its sub-sequence of actions). Finally, a_2, which inherits the attributes from g_1, is related to e_2. Indeed after 1 beat from g_1, we are exactly at the onset of e_2 in respect to the mixed-score timeline. After translating the delay according to e_2, the action a_2 is performed at the detection of e_2.

Hence the sequence, resulted to a run with the ideal input is the following:

$$e_1 \cdot a_1 \cdot (g_1) \cdot 1s \cdot e_2 \cdot a_2$$

where the tempo is 60_{bpm} and (g_1) depicts the activation of the group g_1 not observed in the run since it is an internal system behavior. As expected, the actions a_1 and g_1 are performed simultaneously to the detection of e_1. Then, after the duration of e_1 in physical time (lasting 1 second with 60_{bpm}) the event e_2 is detected and a_2 is performed. ◊

Antescofo Input Events

Antescofo DSL was created to help the composition of mixed-music. One of its main goal is to provide the musical expressivity allowed by the classical western notation. Hence, the list of events defined to specify musicians' events, highlighted in red in the example, contains simple NOTE and CHORD (set of simultaneous NOTE), but also TRILL and MULTI, for specifying respectively the classical trill (tremolo) or a continuous sequence of NOTE (*e.g.* glissandi). An event is defined by three attributes:

1. a set of pitches, being the high of the NOTEs to detect,
2. a duration (a timing variable), and,
3. a label (we suppose for testing a fully labeled score).

Antescofo Output Actions

The list of possible system reactions specified in the DSL contains atomic or compound actions. The former is an elementary action (performed instantaneously after an optional delay) and the latter an encompassed sequence of actions. An action is defined by four attributes:

1. a label,
2. a delay (which is 0 if not explicitly defined),
3. a synchronization strategy *@sync* (inherited from enclosing actions or top-level if not defined), and,
4. an error management strategy *@err* (also inherited).

Notice that we speak of durations for events, because they are seen as pairs of onset with an amount of time until the next event's onset. However, we use the term delays for actions, because we consider the amount of time we have to wait for *performing* the action.

For the following of this section, we only list the actions that are relevant for our testing point of view.

Atomic actions. There are three atomic actions:

1. the assignment of a variable,
2. the emission of a message to an audio application, and,

3. the emission of a kill signal for aborting active compound actions.

The performance of an atomic action results on their instantaneous firing, *i.e.* the assignment, the emission and the termination of actions respectively. Notice that, because they are atomic, we cannot kill such actions which are instantaneous, in the sense of synchronous languages such as Lustre [31] or Esterel [13].

Group of actions. As seen previously in the example, *groups* are compound actions, containing a sub-sequence of locally dependent actions. Groups are used to create musical phrases, and in a sub-sequence, the actions share the same set of attributes by default. When a group is fired, its subsequence of actions is performed concurrently. A group is active from its firing to its termination, either by the performance of its last action or by the reception of a kill signal by itself or one of its ancestor.

For instance, let us consider this following code fragment:

```
1  NOTE C4 1 @name e1
2      group @tight @name ga
3      {
4        1 a2 @name a2
5        1 kill gb @name kgb
6      }
7      0.5 group @name gb
8      {
9        1 a3 @name a3
10       1 kill ga @name kga
11     }
```

The sequence resulted from a run performed following the ideal input sequence with a tempo of 60_{bpm} is:

$$e_1 \cdot (ga) \cdot 0.5s \cdot (gb) \cdot 0.5s \cdot a_2 \cdot 0.5s \cdot a_3 \cdot 0.5s \cdot (kgb)$$

In the example, the group g_b is fired in concurrence with g_a previously fired. A group can thus be seen as a kind of thread launched at its activation. Notice that when a kill signal is fired but none of its targeted action is active, it has no effect and is simply discarded.

Loops. In this work, we only consider one more compound action: the loop[3]. A loop performs a sub-sequence of actions iterated at every P duration, where P is the period, *i.e.* the inverse of the frequency. A loop terminates by the end of the mixed-score interpretation or by the reception of a kill signal by itself or one of its ancestor.

```
1 NOTE C4 1 @name e1
2     loop 1 @tight @name l1
3     {
4         a1 @name a1
5       1 a2 @name a2
6       1 a3 @name a3
7     }
8     2.9 kill l1 @name kl1
```

The run of an ideal input sequence on the mixed-score above is:

$$e_1 \cdot a_1 \cdot 1s \cdot a_1 \cdot a_2 \cdot 1s \cdot a_1 \cdot a_2 \cdot a_3 \cdot 0.9s \cdot (kl1)$$

with $(kl1)$ the internal kill signal terminating all the instances of the *loop* action l_1. Notice that actions can occur simultaneously, in particular, two instances of the same compound action may overlap. However, the ordering of actions is primordial in music, *e.g.* a_1 can start the communication and a_n can send information through the channel opened by a_1.

Hence, the order of the actions considered in the system is the one of the score, in other word, the action's line-code rank. In case of instances of a same compound action, the first created instance has the higher priority. This is enough to entail a strict ordering of actions and, hence, to ensure the determinism.

Sync-Err Attributes

How should the system react when an input is not as expected ?

We can see an unexpected input because:

- its onset was not expected at its detection date (*i.e.* the tempo is not ideal) and the previous event is too long or too short, or
- a previous event was expected but missed during performance.

[3]Notice that such loops are not primitive constructions and can be evaluated using a combination of loop and termination command.

These two cases concern the *synchronization* and the *error management* strategies of the system respectively. Thereafter, we will present two strategies for each case and shows their combined effect on an example with non ideal input traces.

Synchronization. The exclusive attributes *@loose* and *@tight* specify the synchronization of an action. The former, timely synchronizes the action according to the musician's detected tempo only. It results on smooth variations of delay even if a big shift is seen during performance. The latter synchronizes on events and has more impact on delay variations.

Error management. The two exclusive attributes to control the error management are *@local* and *@global*. The first, just does not activate the actions related to a missed event, the other does. Notice that the activation of actions related to a missed event is performed when this event is detected as missed, that is to say, when a next event is detected instead.

```
1  NOTE D5# 1 @name e1
2    group @sync @err
3    {
4              a1
5        0.5   a2
6        0.5   a3
7        0.5   a4
8    }
9
10 NOTE A4 1 @name e2
11   group @sync @err
12   {
13             b1
14       0.5   b2
15   }
```

Example 5.1.29: We depict through examples all the behaviors of two *groups* for the four combinations of the attributes listed previously. In order to clearly present these examples, we describe a run of the above DSL code in a visual representation as shown in Figure 5.5 for the ideal case.

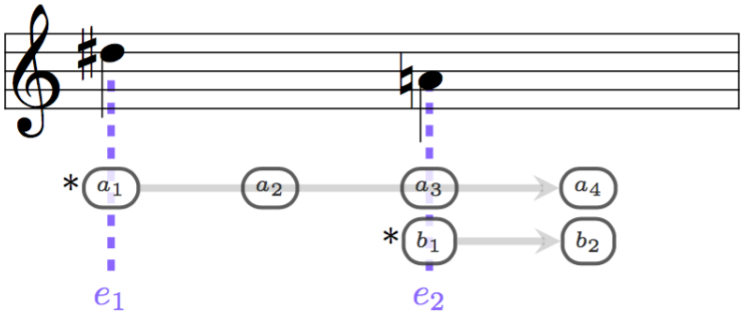

Figure 5.5: An ideal run.

The input events are in the western notation style, their ideal timestamps being depicted with blue dashed lines, terminated by the corresponding event's symbol or "missed". The actions are represented as boxes (that can contain a sequence of simultaneous actions). The two groups are represented horizontally, the top line for the group g_1 and bottom line for the group g_2. Finally, the Antescofo's event-trigger reactions are marked using

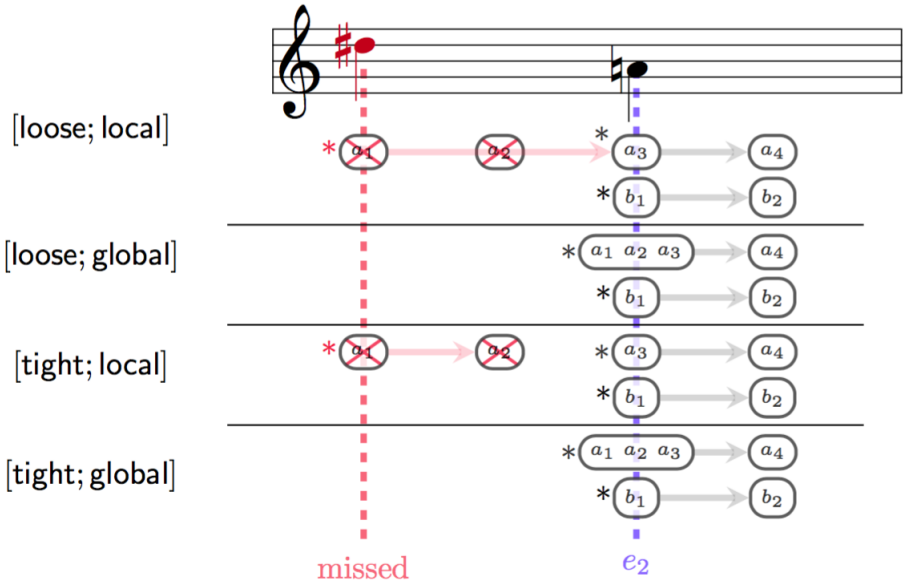

Figure 5.6: A run with e_1 missing.

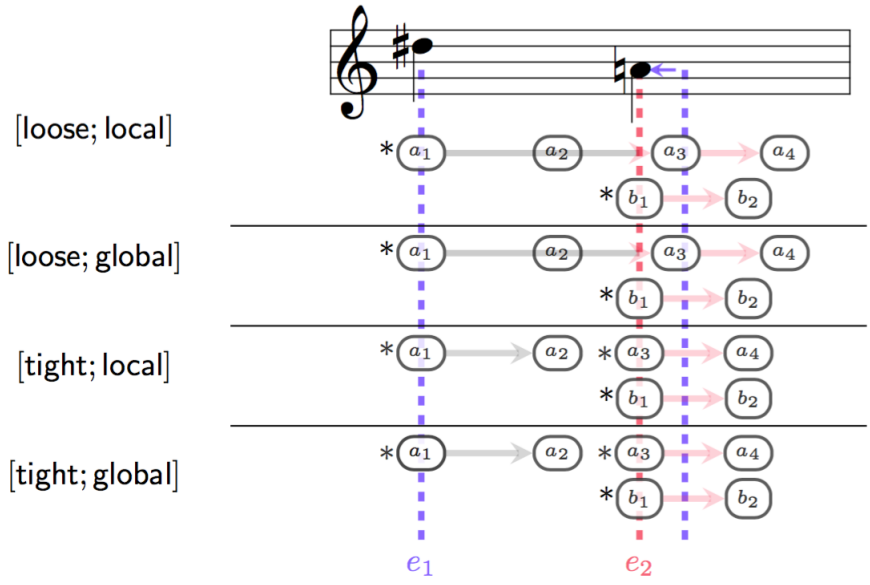

Figure 5.7: A run with e_2 early.

stars, colored in red when the missed detection launches some actions, and time-triggers are denoted with gray arrays, depicted in red if the delay is not ideal.

As expected for the ideal case, after the detection of e_1 the four actions of the first group are triggered 0.5 beat, says second for a tempo of 60_{bpm}, after its previous action. The same behavior is seen for the two actions of the second group with e_2. Notice that no distinction is seen on the ideal case and for all the attribute combinations.

However, as depicted in Figure 5.6, if the first event is missed, a distinction is seen according to the error management attribute, local or global, of the actions. In the local case, the actions a_1 and a_2 are skipped because of the miss of e_1. This is depicted by red crosses on the action boxes. However, the actions a_3 and a_4 are not affected in any cases and are played in the ideal mode because of the good detection of e_2. In the global case, the actions a_1 and a_2 are played when e_1 is detected as missing. It results on the simultaneous emission of a_1, a_2 and a_3.

For tempo variations, an event can be detected earlier than its ideal specification. Depicted in Figure 5.7, if such a case happens, the synchronization attributes specify the reaction of the system. In case of loose attributes, the

delays are ideal until the early detection of e_2. At this point, the arrays become red because the estimated tempo is higher. Indeed, an earlier event implies a tempo acceleration. Thus, the time to wait before a_3 is shorter than the ideal duration. This acceleration has an effect on the delays of the actions a_4 and b_2 too. In case of tight attributes, the action a_3 is not time-triggered and is also sent at the detection of e_2 simultaneously with b_1. The action delays are equally shorter for actions a_2 and b_2, hence, they are sent simultaneously.

Hence, the synchronization strategies played an important role in the coherence of the electronic parts and are difficult to deduce for non trivial input performances. As an example here, it results in playing simultaneously a_3 and a_4 with b_1 and b_2 or not.

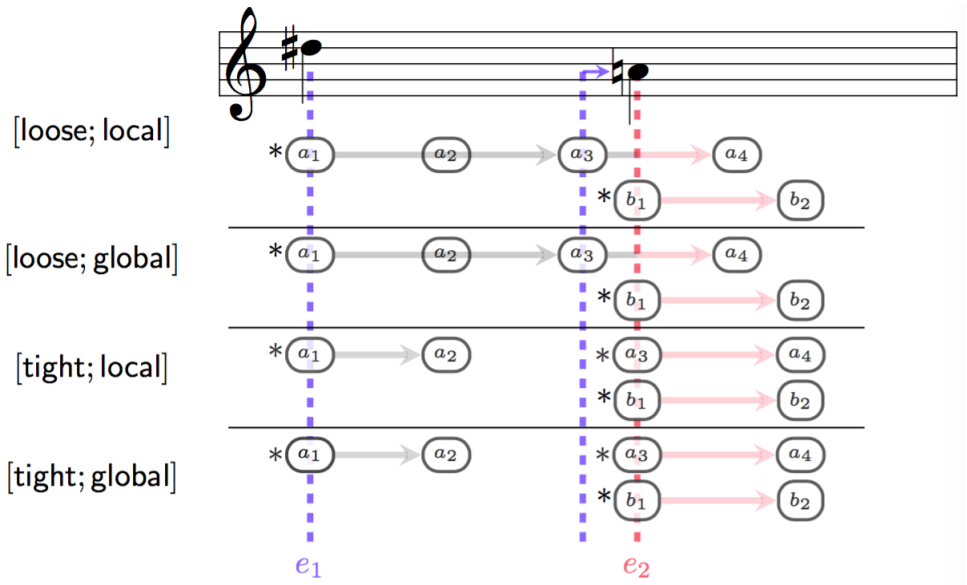

Figure 5.8: A run with e_2 late.

Finally, Figure 5.8 depicts the late occurrence of event. We can deduce the same conclusion as the earlier case, but here the action a_3 is sent before the event e_2 or at the same time. The delays waited for a_4 and b_2 become longer than the ideal duration because a later event decreases the estimated tempo. ◊

5.2 Model-Based Testing Antescofo

Antescofo is a complex real-time system and is involved in a non-deterministic input environment. However, the system must react to these inputs in a strictly specified manner. Hence, lots of errors are possible.

We applied our TMBT framework to Antescofo, in order to ensure a good execution of the IMS and prevent from a misbehavior during concerts.

To do such formal tests, we use as the requirements in Chapter 4, the *mixed-score* required by Antescofo, which timely specifies the musicians input events traces and the related Antescofo output reactions. In this chapter:

- the abstract syntax of Antescofo requirements and its construction rules are presented in Section 5.2.1,
- then, we depict in Section 5.2.2 some IRTMs constructed by the previous rules, and,
- finally, the implemented steps of our test framework, are detailed in Section 5.2.3.

We recall and depict in Figure 5.9 the overview of our implemented testing procedures. On the left, the online approach is detailed, aside of its offline counterpart depicted on the right. Both of these testing approaches start from an Antescofo mixed-score and construct automatically the corresponding IRTM (1):

- Then, on the right, the offline approach generates a set of input traces σ_{in} from the model and/or the mixed-score information (2). Once some input traces have been generated, the model is used to compute the corresponding reference traces σ_{ref} by simulation (3). Then, the input traces are sent to Antescofo (4) in order to deduce the monitored traces σ_{moni}. Finally, the reference traces are compared to the monitored ones resulting in a verdict (5).
- On the left, the online approach uses the Virtual Machine (VM) to execute the model. Input test data is generated on the fly using an adapter (*Adt*). The generated inputs are also sent to Antescofo, as the input of artificial musicians. The monitored outputs of the system are compared online to the reference trace (5), event by event. An error is reported if some Antescofo reactions are not expected or missed in respect to the model.

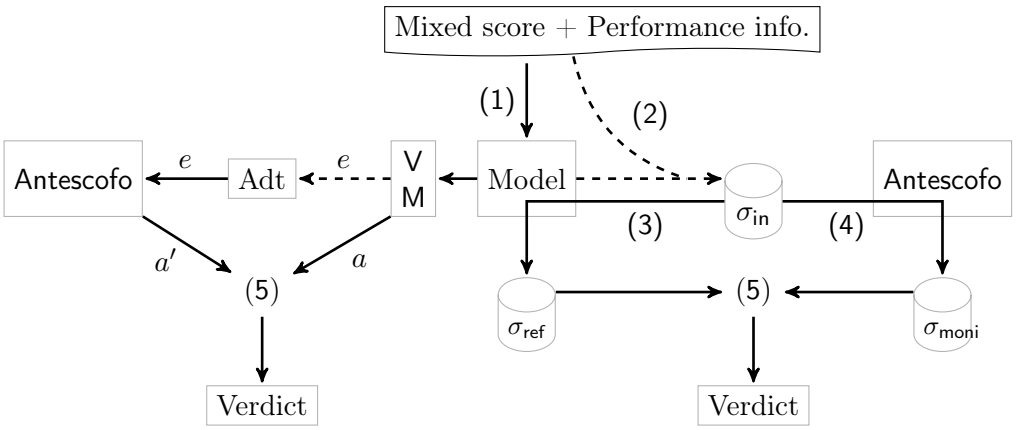

Figure 5.9: Two implementations of score-based IMS testing procedures: on the left online method - on the right offline method.

5.2.1 Model Construction

Compiling mixed scores into IRTMs has been implemented as a command line tool, called automata, written in C++ on the top of the original Antescofo's parser. In order to implement the construction rules accordingly, the abstract syntax given as input of our TMBT framework is the result of the system's parser on the subset of Antescofo syntax presented in Section 5.1.

Abstract Syntax

The abstract syntax is over the alphabets *Evt* and *Act* defining the set of event and action symbols respectively. In order to express synchronization, error management strategies and loops in the model, we extend the abstract syntax of Section 4.1.1 with Antescofo attributes given the syntax depicted in Figure 5.10.

A mixed score is a finite sequence of events of the form $\mathsf{evt}(e, d, as)$ with $e \in Evt$, $d \in \mathbb{R}_{\geq 0}$ the duration of e, and as the top-level group triggered by e. A duration can be in performance time (b) or physical time (s). An action can be of the form:

. $\mathsf{act}(d, a, al)$ with $a \in Act$ an atomic action,
. $\mathsf{act}(d, as, al)$, with as a finite sequence of actions, called a group, or
. $\mathsf{act}(d, s, al, p)$ where $p \in \mathbb{R}_{>0}$ is the period duration, in performance or physical time, of the loop.

score	::=	ε | event score	
event	::=	evt(e, dur, seq)	$e \in Evt$
seq	::=	ε | action seq	
action	::=	act(dur, a, al) | act(dur, seq, al) | act(dur, seq, al, dur)	$a \in Act$
dur	::=	d timeUnit	$d \in \mathbb{R}$
timeUnit	::=	s | b	
al	::=	sync? | err?	
sync	::=	loose | tight	
err	::=	local | global	

Figure 5.10: Grammar of the abstract syntax for the tested fragment of Antescofo DSL.

For all the cases, $d \in \mathbb{R}_{\geq 0}$ is the delay (in performance or physical time) to wait for before performing the action, and al is a list of attributes containing the synchronization strategy sync and the error management strategy err information. The default values of these attributes are [loose, global].

Construction Rules

The construction rules are based on the abstract syntax defined in Figure 5.10. The construction consists in a single traversal of the AST which returns a model composed of: $\mathcal{M}_{\text{env}}$, a test environment as described in Section 4.3.1 and $\mathcal{M}_{\text{sys}}$, the Antescofo specification.

The rule $\vdash_{\text{all}}$ constructs the FSM associated to a mixed score according to two cases:

$$\frac{}{: \emptyset \vdash_{\text{all}} \mathcal{A}_\emptyset} \qquad \frac{: ms \vdash_{\text{env}} \mathcal{M}_{\text{env}} \qquad : ms \vdash_{\text{proxy}} \mathcal{P} \qquad : ms \vdash_{\text{sys}} \mathcal{A}}{: ms \vdash_{\text{all}} \mathcal{M}_{\text{env}} \| \mathcal{P} \| \mathcal{A}}$$

If the score is empty, the rule $\vdash_{\text{all}}$ returns an empty FSM $\mathcal{A}_\emptyset$, containing an empty set of locations. Otherwise, it applies to the mixed-score ms three other rules:

$\vdash_{\text{env}}$ for constructing the test environment model $\mathcal{M}_{\text{env}}$,

$\vdash_{\text{proxy}}$ to construct the proxy $\mathcal{P}$, the interface between $\mathcal{M}_{\text{env}}$ and $\mathcal{A}$,

$\vdash_{\text{sys}}$ in charge of creating $\mathcal{A}$, the specification of Antescofo.

where $\mathcal{M}_{\mathsf{env}}$, $\mathcal{P}$ and $\mathcal{A}$ have the type $\langle 1, 0\rangle$. Moreover, we let $\mathcal{M} = \mathcal{M}_{\mathsf{env}} \| \mathcal{M}_{\mathsf{sys}}$ with $\mathcal{M}_{\mathsf{sys}} = \mathcal{P} \| \mathcal{A}$, the complete model $\mathcal{M}$ is a parallelization of the environment $\mathcal{M}_{\mathsf{env}}$ and Antescofo model $\mathcal{M}_{\mathsf{sys}}$ which is itself a parallel composition of the proxy $\mathcal{P}$ and $\mathcal{A}$.

Proxy FSM. A FSM of the form $\mathcal{P} = \langle Evt, Sig, \mathcal{L}^{\mathcal{P}}, \ell_0^{\mathcal{P}}, \Delta^{\mathcal{P}}\rangle$, called proxy is in charge of receiving detected events and signaling to the other FSMs the missing events, depicted using signals of the form $\overline{e}$. Here, we define an error as a missing event e_i, detected at the arrival of a next event e_{i+k}, with $k > 0$. The proxy FSM approach is modular and permits to replace this definition of error with alternative ones, without changing the rest of the FSM. We present here a construction of $\mathcal{P}$ for $n_{err} = 1$:

$$\frac{: \mathsf{evt}(e, d, as) \vdash_{\mathsf{pevt}_0} \mathcal{P}_0^{\langle 1,2\rangle} \qquad e : ms' \vdash_{\mathsf{proxy}_1} \mathcal{P}^{\langle 2,0\rangle}}{: \mathsf{evt}(e, d, as)::ms' \vdash_{\mathsf{proxy}} \mathcal{P}_0^{\langle 1,2\rangle} + \mathcal{P}^{\langle 2,0\rangle}}$$

$$\frac{e_{i-1} : \mathsf{evt}(e_i, d, as) \vdash_{\mathsf{pevt}_1} \mathcal{P}_1^{\langle 2,2\rangle} \qquad e_i : ms' \vdash_{\mathsf{proxy}_1} \mathcal{P}^{\langle 2,0\rangle}}{e_{i-1} : \mathsf{evt}(e_i, d, as)::ms' \vdash_{\mathsf{proxy}_1} \mathcal{P}_1^{\langle 2,2\rangle} + \mathcal{P}^{\langle 2,0\rangle}}$$

$$\frac{}{e : \emptyset \vdash_{\mathsf{proxy}_1} \mathcal{F}^{\langle 2,0\rangle}}$$

The rule $\vdash_{\mathsf{pevt}_0}$ initializes the FSM $\mathcal{P}$, by waiting for the first event $\mathsf{evt}(e, d, as)$. The rule $\vdash_{\mathsf{pevt}_1}$ treats each following event e_i of the score. The FSMs constructed by these rules are depicted in Figure 5.11. Provider and seeker 1 correspond to the case when this event e_i is received after the previous event e_{i-1}, while provider and seeker 2 correspond to the case when e_i is received whereas e_{i-1} was not received. In the latter case, the signal $\overline{e_{i-1}}$ is emitted to notify that the last event e_{i-1} is missing.

In practice, we have implemented a proxy for $n_{err} <= 7$, because Antescofo assumes that no more than 7 events can be consecutively missed.

FSM for the score reactions. The rule $\vdash_{\mathsf{sys}}$ constructs a FSM of the form $\mathcal{A} = \langle Evt \cup Sig, Act, \mathcal{L}^{\mathcal{A}}, \ell_0^{\mathcal{A}}, \Delta^{\mathcal{A}}\rangle$ specifying the behavior of the system in reaction to the events of the environment, *i.e.* the automatic accompaniment.

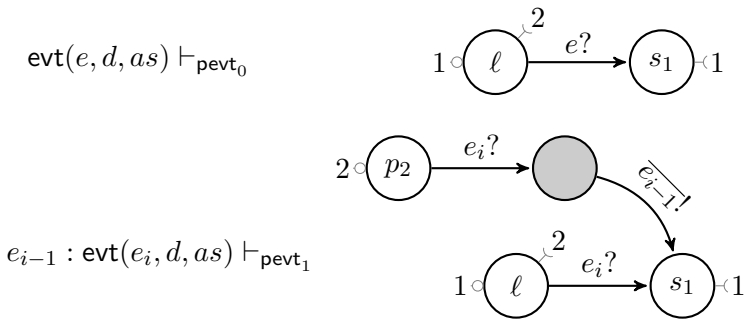

Figure 5.11: The parts of the proxy FSM.

$$\frac{0, \mathsf{evt}(e',d',\_)::ms', \mathsf{evt}(e,d,as), \bot : as \vdash^{\mathsf{loose,global}}_{\mathsf{seq}} \mathcal{A}_s^{\langle 2,0\rangle} \quad : \mathsf{evt}(e',d',\_)::ms' \vdash_{\mathsf{sys}} \mathcal{A}^{\langle 1,0\rangle}}{: \mathsf{evt}(e,d,as)::\mathsf{evt}(e',d',\_)::ms' \vdash_{\mathsf{sys}} (\mathcal{T}_e^{\langle 1,2\rangle} + \mathcal{A}_s^{\langle 2,0\rangle}) \| \mathcal{A}^{\langle 1,0\rangle}}$$

$$\frac{: ms' \vdash_{\mathsf{sys}} \mathcal{A}^{\langle 1,0\rangle}}{: \mathsf{evt}(e,d,\varepsilon)::ms' \vdash_{\mathsf{sys}} \mathcal{A}^{\langle 1,0\rangle}} \qquad \frac{}{: \emptyset \vdash_{\mathsf{sys}} \mathcal{F}^{\langle 1,0\rangle}}$$

The first case of $\vdash_{\mathsf{sys}}$, on the top, returns a FSM associated to $\mathsf{evt}(e,d,as)$ and describes the behavior of a top-level group, triggered by an event e, and containing the sequence of actions as. This part is the sequential composition of $\mathcal{T}_e$, a trigger FSM labeled by e and the FSM associated to as, constructed by a call to the rule $\vdash^{\mathsf{loose,global}}_{\mathsf{seq}}$. This rule, presented below, treats as as a group with attributes loose, global, which is the behavior defined for top-groups. This part is composed in parallel with the FSM $\mathcal{A}$ built by a recursive call of $\vdash_{\mathsf{sys}}$ on $\mathsf{evt}(e',d',\_)::ms'$, the rest of the score. When the end of score is reached, right bottom case, the final FSM is constructed by adding an ender $\mathcal{F}^{\langle 1,0\rangle}$ with 1 provider. Finally, the last case, on the bottom left, passes to the next event if no action is related to e.

FSM for actions sequences. We now define the rule $\vdash^{al}_{\mathsf{seq}}$ for building the FSM associated to an action sequence as, under the attributes in al. The rule will parse the sequence as and build a FSM that will send these actions according to the strategies in al. This rule will also traverses the list of events occurring in the score, after the event e. Indeed, synchronization with these events is required in some strategies.

Moreover, the event durations d_e and the action delays d may be in a different time unit. We recall that $\Phi(d)$ is the evaluation function of a duration from a relative time into physical time. This function, has no effect if the duration is already in physical time, and uses the ideal tempo curve $f^{\sigma_{\mathsf{in}}}$ defined in the score for translating the durations $d^{\sigma_{\mathsf{in}}}$ from performance time unit.

Every call to this rule will have the form

$$\delta, ms, evt, x : as \vdash^{al}_{\mathsf{seq}} \mathcal{A}$$

where the auxiliary argument:

- δ^{phy} is an accumulator in physical time. It is the sum of the actions' delays parsed so far in the given sequence, minus the durations of the processed events. In other term, it is the duration between the *closest event* and the action itself.

- ms is the list of events that remain to be processed, its first event being the next event to detect, following the score,

- evt called *closest event*, is the last event before the action currently parsed, *i.e.* the first action of as, in the timeline defined by the score,

- x is a flag whose role is explained later.

Then, we define the possible cases during the traversal of a sequence of actions as. The base case, is when the list of actions is empty. It simply returns an ender.

$$\frac{}{\delta, ms, evt, x : \varepsilon \vdash^{al}_{\mathsf{seq}} \mathcal{F}^{\langle i,0 \rangle}}$$

where i depends on the attribute sequence al.

When the list of actions is not empty, a call to $\vdash^{al}_{\mathsf{seq}}$ will first update the accumulator δ by adding the delay d of the currently parsed action, and carry on with a call to a second rule $\vdash^{al}_{\mathsf{seq}_1}$.

$$\frac{\delta + \Phi(d), ms, evt, \bot : \mathsf{act}(d, a, al')::as' \vdash^{al}_{\mathsf{seq}_1} \mathcal{A}}{\delta, ms, evt, \bot : \mathsf{act}(d, a, al')::as' \vdash^{al}_{\mathsf{seq}} \mathcal{A}}$$

Note that the flag x must be $\bot$ and keeps this value. The rule $\vdash^{al}_{\mathsf{seq}_1}$ will look for the *closest event* before the action currently parsed, in order to update the third auxiliary argument.

$$\frac{\delta - \Phi(d_e), ms', \mathsf{evt}(d', e', as'_e), \top : as \vdash^{al}_{\mathsf{seq}_1} \mathcal{A}}{\delta, \mathsf{evt}(d', e', as'_e)::ms', \mathsf{evt}(d_e, e, as_e), x : as \vdash^{al}_{\mathsf{seq}_1} \mathcal{A}}$$

if $\delta \geq \Phi(d_e)$.
Concretely, the rule $\vdash^{al}_{\mathsf{seq}_1}$ is applied if the actions' delay accumulator δ is greater than the duration of the closest event d_e, *i.e.* if the *closest event* $\mathsf{evt}(d_e, e, as_e)$ finishes before the action currently parsed. Then, the third auxiliary argument is updated to $\mathsf{evt}(d', e', as'_e)$, the head of the secondary argument, which is removed from the list and the flag, the fourth auxiliary argument, is set to $\top$. Moreover, the duration d_e of the closest event e is subtracted from the accumulator δ to compute the rest of the delay to wait for until the new *closest event*.

Otherwise, if d_e finishes after the action currently parsed, it means that we have found the good *closest event* before this action. Then, we can process by sending the current action. We consider three cases. The first case is for an *atomic action* $\mathsf{act}(d, a, al')$, with $a \in Act$:

$$\frac{\begin{array}{c} d, \delta, e', e, x : \vdash^{al}_{\mathsf{delay}} \mathcal{A}_d^{\langle n,m\rangle} \qquad \mathsf{act}(d, a, al') \vdash^{al}_{\mathsf{atom}} \mathcal{A}_a^{\langle m,m\rangle} \\ \delta, \mathsf{evt}(d', e', as'_e)::ms', \mathsf{evt}(d_e, e, as_e), \bot : as' \vdash^{al}_{\mathsf{seq}} \mathcal{A}^{\langle m,0\rangle} \end{array}}{\delta, \mathsf{evt}(d', e', as'_e)::ms', \mathsf{evt}(d_e, e, as_e), x : \mathsf{act}(d, a, al')::as' \vdash^{al}_{\mathsf{seq}_1} \mathcal{A}_d^{\langle n,m\rangle} + \mathcal{A}_a^{\langle m,m\rangle} + \mathcal{A}^{\langle m,0\rangle}}$$

if $\delta < \Phi(d_e)$.
In order to treat $\vdash^{al}_{\mathsf{seq}}$ with an atomic action, we call first $\vdash^{al}_{\mathsf{delay}}$ to specify the management of the delay, when $d > 0$, according to the attribute list al. The FSM $\mathcal{A}_d$, returned by $\vdash^{al}_{\mathsf{delay}}$, is concatenated with $\mathcal{A}_a$, a FSM in charge of sending the action a. Both $\mathcal{A}_d$ and $\mathcal{A}_a$ will be defined below according to the attribute list al. Finally, we call $\vdash^{al}_{\mathsf{seq}}$ to iterate on the rest of the action sequence as' and concatenate the result to the FSM already computed. Note that the flag is set to $\bot$ in this recursive call.

The case of a *compound action* $\mathsf{act}(d, as_a, al')$ is as follows:

$$\frac{\begin{array}{c} d, \delta, e', e, x : \vdash^{al}_{\mathsf{delay}} \mathcal{A}_d^{\langle n,m\rangle} \\ \delta, \mathsf{evt}(d', e', as'_e)::ms', \mathsf{evt}(d_e, e, as_e), \bot : as_a \vdash^{al'}_{\mathsf{seq}} \mathcal{A}_{sa}^{\langle m',0\rangle} \\ \delta, \mathsf{evt}(d', e', as'_e)::ms', \mathsf{evt}(d_e, e, as_e), \bot : as' \vdash^{al}_{\mathsf{seq}} \mathcal{A}^{\langle m,0\rangle} \end{array}}{\delta, \mathsf{evt}(d', e', as'_e)::ms', \mathsf{evt}(d_e, e, as_e), x : \mathsf{act}(d, as_a, al')::as' \vdash^{al}_{\mathsf{seq}_1} \mathcal{A}}$$

if $\delta < \Phi(d_e)$,

$$\text{where } \mathcal{A} = \mathcal{A}_d^{\langle n,m\rangle} + (\mathcal{I}^{\langle m,m\rangle} \| \mathcal{A}_{sa}^{\langle m',0\rangle}) + \mathcal{A}^{\langle m,0\rangle} \text{ if } m \geq m',$$
$$\text{and } \mathcal{A} = \mathcal{A}_d^{\langle n,m\rangle} + (\mathcal{I}^{\langle m,m\rangle} \| \mathcal{I}^{\langle 2,2\rangle} + \mathcal{A}_{sa}^{\langle m',0\rangle}) + \mathcal{A}^{\langle m,0\rangle} \text{ otherwise.}$$

The only difference with the case of an atomic action is the treatment of the sub sequence of actions itself, which is processed with a recursive call to $\vdash_{\mathsf{seq}}^{al'}$, applied to the sequence of actions as_a, the content of the compound action, and following the attributes al'.

The two operations below the rules prevent from having more connectors to fire than those of the parent FSM. Indeed, this parallelization allows to send a sub-FSM according to the mode of its parent one. Hence, an erroneous mode will launch a sub-FSM in an erroneous mode. However, a problem arises if the parent has less connectors than the sub-FSM, therefore, we cast the children's providers to two connectors with $\mathcal{I}^{\langle 2,2\rangle}$ to avoid this case.

It remains to consider where the second auxiliary argument is empty, because we have reached the end of the event list on the score. In this case, the sequence of actions is treated with the loose strategy, whatever the strategy specified in the score. The case of an *atomic action* is then:

$$\frac{\mathsf{act}(d,a,al') \vdash_{\mathsf{atom}}^{\mathsf{loose},err} \mathcal{A}_a^{\langle m,m\rangle} \qquad \begin{array}{c} d,\delta,e,e,\bot : \vdash_{\mathsf{delay}}^{\mathsf{loose},err} \mathcal{A}_d^{\langle n,m\rangle} \\ \delta,\varepsilon,evt,\bot : as' \vdash_{\mathsf{seq}}^{sync,err} \mathcal{A}^{\langle m,0\rangle} \end{array}}{\delta,\varepsilon,evt,x : \mathsf{act}(d,a,al')::as' \vdash_{\mathsf{seq}_1}^{sync,err} \mathcal{A}_d^{\langle n,m\rangle} + \mathcal{A}_a^{\langle m,m\rangle} + \mathcal{A}^{\langle m,0\rangle}}$$

And the case of a compound action is treated similarly as above.

FSM for loops. The last case is for a *loop action* $\mathsf{act}(d, sa_a, al', p)$. We create for each loop a signal $\lambda \in Sig$, called *kill signal*, terminating the sub-sequences related to a loop.

$$\frac{\begin{array}{c} d,\delta,e',e,x : \vdash_{\mathsf{delay}}^{al} \mathcal{A}_d^{\langle n,m\rangle} \qquad p,\lambda : \vdash_{\mathsf{loop}}{}^{\langle \ell,k\rangle}\mathcal{A}_\ell^{\langle m,m\rangle} \\ \lambda,\delta,\mathsf{evt}(d',e',as'_e)::ms',\mathsf{evt}(d_e,e,as_e),\bot : as_a \vdash_{\mathsf{seq}}^{al'}{}^{k}\mathcal{A}_{sa}^{\langle m',0\rangle} \\ \delta,\mathsf{evt}(d',e',as'_e)::ms',\mathsf{evt}(d_e,e,as_e),\bot : as' \vdash_{\mathsf{seq}}^{al} \mathcal{A}^{\langle m,0\rangle} \end{array}}{\delta,\mathsf{evt}(d',e',as'_e)::ms',\mathsf{evt}(d_e,e,as_e),x : \mathsf{act}(d,sa_a,al',p)::as' \vdash_{\mathsf{seq}} \mathcal{A}_d^{\langle n,m\rangle}+\mathcal{A}_1+\mathcal{A}_2+\mathcal{A}^{\langle m,0\rangle}}$$

$$\text{where } \mathcal{A}_1 = (\mathcal{I}^{\langle m,m\rangle} \|^k \mathcal{I}^{\langle m',m'\rangle} + {}^k\mathcal{A}_{sa}^{\langle m',0\rangle}) \text{ if } m \geq m',$$
$$\text{and } \mathcal{A}_1 = (\mathcal{I}^{\langle m,m\rangle} \|^k \mathcal{I}^{\langle 2,2\rangle} + {}^k\mathcal{A}_{sa}^{\langle m',0\rangle}) \text{ otherwise,}$$
$$\text{and } \mathcal{A}_2 = (\mathcal{I}^{\langle m,m\rangle} \| ({}^{\langle \ell,k\rangle}\mathcal{A}_\ell^{\langle m,m\rangle} + ({}^{\langle m,\ell\rangle}\mathcal{L} \|^k \mathcal{A}_{sa}^{\langle m',0\rangle}))).$$

The rule $\vdash_{\mathsf{seq}}$ for loops follows the third construction rules (concurrent reaction) described in Section 4.2.4 but for FSMs of type m. We first call $\vdash^{al}_{\mathsf{delay}}$ for the management of the delay and then launch the FSM related to as_a, the content of the compound action. Then, $\vdash_{\mathsf{loop}}$ initializes a loop for each provider in m and launches the same FSM every p duration. Each linker $^{\langle m,\ell\rangle}\mathcal{L}$ merges the i^{th} first location of a loop with a target of the next *and*-transition as presented in Section 4.2.4 for provider 1. The FSM constructed by the rule $\vdash_{\mathsf{loop}}$ is depicted in Figure 5.12 for $m = 1$.

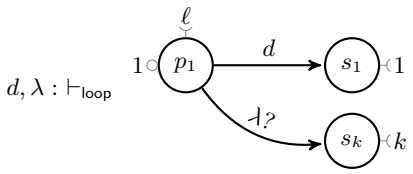

Figure 5.12: FSM constructed by $\vdash_{\mathsf{loop}}$.

FSM for action's delays and atomic actions. The attribute list will determine how to manage a delay d in the call of the rule $\vdash^{al}_{\mathsf{delay}}$ and how to treat an atomic action $a \in Act$ in the call of the rule $\vdash^{al}_{\mathsf{atom}}$. We detail in the following of the section the FSMs constructed for every combination of attributes al.

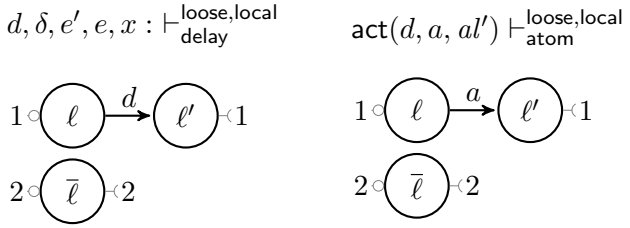

Figure 5.13: FSM managing the delay d and the atomic action a for the attributes loose, local.

Case $al = \mathsf{loose}, \mathsf{local}$**.** In this case, depicted in Figure 5.13, the FSM built by $\vdash_{\mathsf{delay}}$ waits for the delay d when in provider 1 (normal mode) and the FSM of $\vdash_{\mathsf{atom}}$ sends the action a, with no care in event detections. In the error mode, provider 2, both $\vdash_{\mathsf{delay}}$ and $\vdash_{\mathsf{atom}}$ do nothing, the delay and action are skipped.

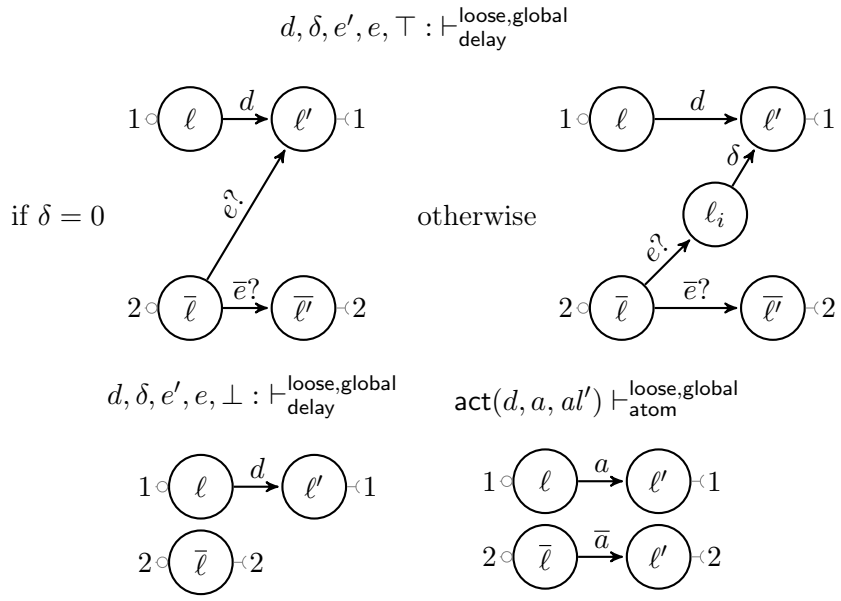

Figure 5.14: FSMs managing the delay d and the atomic action a for the attributes loose, global.

Case $al = \mathsf{loose}, \mathsf{global}$**.** Figure 5.14 depicts the FSM constructed by $\vdash_{\mathsf{delay}}$ and $\vdash_{\mathsf{atom}}$ for the combination of attributes loose and global, with different cases according to the flag and the value of the accumulator δ.

On the left part of the Figure 5.14, the FSM waits for d time units in normal mode (provider 1) and does not wait in error mode (provider 2). On the top right part, the expected *closest event* before the current action is detected, and causes a transition from the error mode into the normal mode. The duration δ is the delay between e and the action, computed in the accumulator of $\vdash_{\mathsf{seq}_1}$. For the treatment of an atomic action (on the bottom and right part), the action is sent into the normal mode. However, for the error mode the emission depends on its proper attribute (al'), depicted with the notation $\overline{a}$, it is sent for the attribute global and not otherwise.

Case $al =$ tight, _**.** The case of the attribute tight is depicted in Figure 5.15 for the rule $\vdash_{\mathsf{delay}}$. In the first case on the left, when the flag is $\bot$, the FSM waits, when in normal mode – provider 1, for the delay d before the current action. Recall that the third auxiliary argument e' is the event, next to the *closest event* e, the fourth auxiliary argument of rules $\vdash_{\mathsf{seq}}$. If this

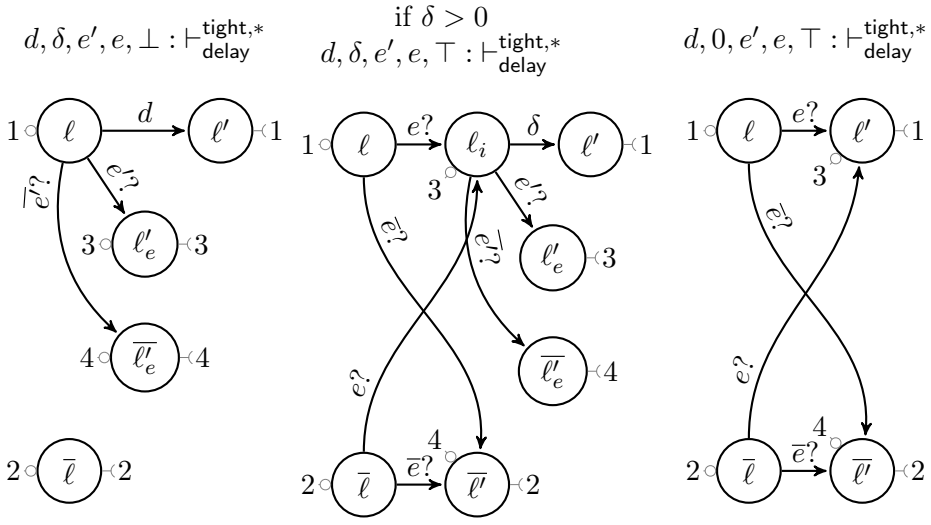

Figure 5.15: FSM managing the delay for the attribute tight.

event e' arrives earlier than expected, *i.e.* before d, then the FSM switches from normal mode (provider 1) to another mode called *early mode* (provider 3). If e' is notified as missing (signal $\overline{e'}$) before e' was expected, then the FSM switches from normal mode (provider 1) to a fourth mode called *early error mode* (provider 4). The provider 2 corresponds to the error mode, as above.

In the second case (on the middle), when the flag is $\top$, the FSM synchronizes the current action a to the *closest event* e, *i.e.* it waits first for the event e, transition from the provider 1 – normal mode, and then waits for δ, the delay, computed by $\vdash_{\mathsf{seq}_1}$, between e and the current action. If, instead of receiving e, the FSM receives a notification that e is missing (signal $\overline{e}$) then it moves to the error mode (provider 2). Moreover, if the next event e' arrives (resp. is detected as missing) earlier than expected, then there is a move to the early mode – provider 3 (resp. the early error mode – provider 4).

In the third case on the right, the delay δ is null, hence it is just skipped.

Note that the composition of such FSMs with 4 providers and 4 seekers is managed properly by the above rules, using appropriate idlers $\mathcal{I}^{\langle m,m\rangle}$ and enders $\mathcal{F}^{\langle m,0\rangle}$, with $m = 2$ or 4, for a correct type inference.

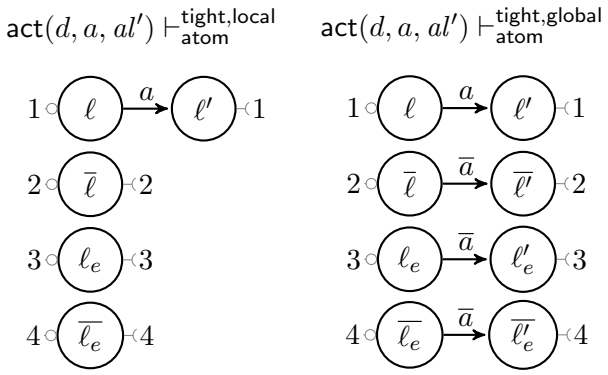

Figure 5.16: FSMs managing atomic action for the attributes tight, local (left) and tight, global (right).

Case $al =$ tight, local**.** The case of the combination of the attributes tight and local for the rule $\vdash_{\mathsf{atom}}$ is depicted in Figure 5.16 (left). The local strategy simply skips the action a when in early, early error or error modes (provider 3, 4, 2 respectively).

Case $al =$ tight, global**.** The case of the combination of the tight and global attributes is depicted in Figure 5.15 for the rule $\vdash_{\mathsf{delay}}$ and in Figure 5.16 for the rule $\vdash_{\mathsf{atom}}$.

The only difference with the case tight and local is for the error mode and the early detection of next events. If the second happens, all the not yet handled actions are sent directly, until this next event, and not skipped as in the previous case.

5.2.2 Antescofo Models

Model representations are an effective approach to clearly depict system behaviors. We provided in our framework several model formats including: a byte-code listing, a graphical representation of IRTMs using the graphviz tool, a Uppaal network of Timed Automata (.xta), and, a Uppaal model with its graphical extension (.ugi).

We present in this section some models returned by our framework after the application of the construction rules in Section 5.2.2 to the example described in Section 5.1, recalled below.

```
1  NOTE D5# 1 @name e1
2    group @sync @err
3    {
4              a1
5        0.5   a2
6        0.5   a3
7        0.5   a4
8    }
9
10 NOTE A4 1 @name e2
11   group @sync @err
12   {
13             b1
14       0.5   b2
15   }
```

IRTM representation. We depict the byte-code listing of the example for the attributes loose, local in the Appendices A. Its corresponding graphviz model is depicted in Figure 5.17 and allows a better view of the model.

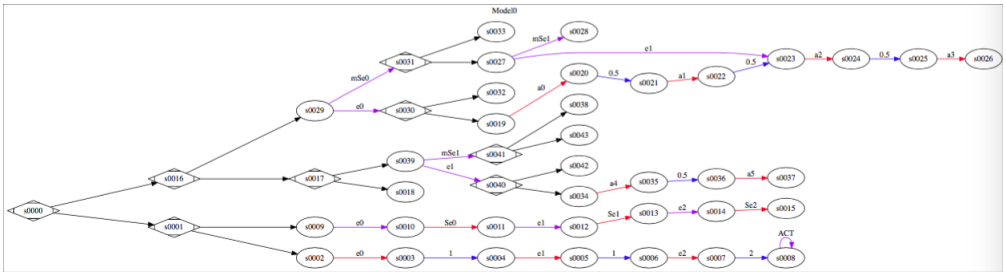

Figure 5.17: Graphviz representation of the IRTM for attributes loose, local.

Uppaal TA. In case of Uppaal representations, the model corresponding to the example for the attributes tight, global is depicted Figures 5.18 and 5.19. The Uppaal representation may imply a coordinate computation of Uppaal model items during compilation to see TA models in a human-understandable fashion. It greatly improved the validation of our whole models by the developers of Antescofo.

Figures 5.18 depicts a single tight global group in Uppaal. Normal or error

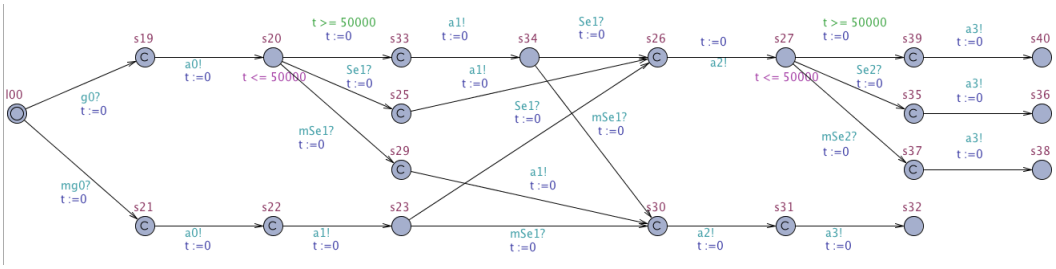

Figure 5.18: A Uppaal model of the example for attributes tight, global.

modes and the possible early modes are easily understandable. Simulation of the whole network is depicted in Figure 5.19, checking if the model is well formed.

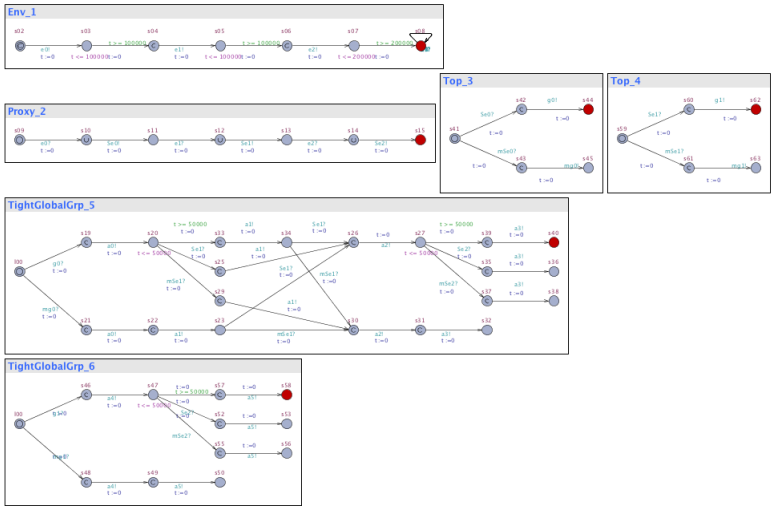

Figure 5.19: A Uppaal simulation view with the network for attributes tight, global.

5.2.3 Applying Test Framework

Now that we can construct automatically a IRTM or its corresponding Uppaal network, the other steps of our TMBT framework can be presented.

Given a model $\mathcal{M}$ constructed with the construction rules presented in

Section 5.2.1, this section details the implementation of our TMBT framework in oder to apply the testing procedure to the score-based IMS Antescofo. We implemented the two offline and online approaches in our framework, both the approaches include the preliminary phase of the compilation from mixed-scores into IRTM (Section 5.2.1). We have developed testing solutions based on existing tools, but also developed our own tools, better suited to our case study.

Compilation of Model. Compiling mixed scores into IRTMs has been implemented as a command line tool, written in C++ on the top of the original Antescofo's parser. The parsing produces an Abstract Syntax Tree which is traversed using a visitor pattern in order to build the IRTM following the approach presented in Section 4.2. Several options are offered for the construction of IRTMs related to the environment $\mathcal{M}_{\mathsf{env}}$. In particular to fix the values of n_{err} and κ from the Section 4.2. The most general case, any note can be missed, results in a model $\mathcal{M}_{\mathsf{env}}$ with a quadratic number (in score's size) of transitions and an exponential number of possible input traces. The explosion can be controlled by choosing appropriate hypotheses on the environment $\mathcal{M}_{\mathsf{env}}$. The software has been newly designed to follow the formal specifications of the construction rules presented in Section 5.2.2.

Offline Testing Approach

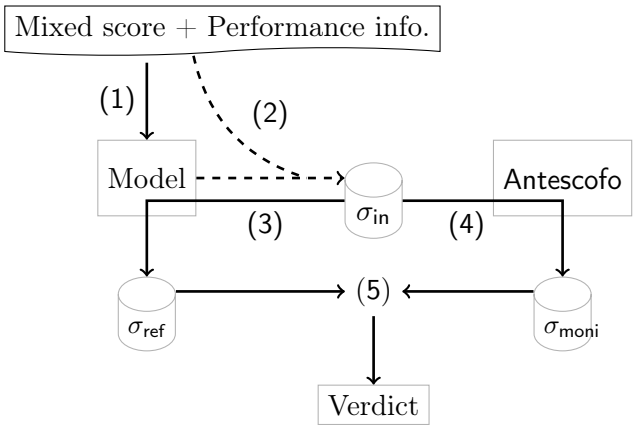

Figure 5.20: Offline Score-based IMS testing procedures.

Figure 5.20 outlines the implementation of our TMBT framework with

an offline generation of input traces. This workflow, following the principles presented in Section 4.1, proceeds in several steps described below.

The generation, simulation and comparison steps have been already presented in the manuscript. More precisely, once the model $\mathcal{M}$ is constructed:

- We generate a set of input trace $\mathcal{T}_{\mathsf{in}}$ following the generation step, presented in Section 4.1.3 and using one of the algorithms detailed in Section 4.3.

  We have considered a third alternative for the generation of test input traces, based on an audio recording. The developers of the IMS Antescofo use to work with sound files in order *e.g.* to analyse a specific performance that causes errors. Such sound files can be translated into input traces simply by marking the timestamps of their event's onsets. We can do that manually or with a software, *e.g.* Antescofo itself, which can trace the events triggered when the listening machine detects them from the audio file.

- We follow the simulation step presented in Section 4.1.4. Hence, the command tool Verifyta or CoVer is used to simulate a Uppaal network, and the VM is used to simulate IRTMs.

- Comparison is performed following the algorithm in Section 4.1.6 for offline approaches.

Translation into Uppaal Model

In a first step, after the construction of the IRTMs $\mathcal{M}_{\mathsf{env}}$ and $\mathcal{M}_{\mathsf{sys}}$ from the given mixed score, using the techniques and tools presented in Sections 5.2.2, these IRTMs are translated into TA networks, respectively $\mathcal{M}'_{\mathsf{env}}$ and $\mathcal{M}'_{\mathsf{sys}}$, as described in Section 3.2. Consequently, this approach works under the restrictions $\mathsf{R}_1 - \mathsf{R}_3$ needed in Section 3.2.

A problem appears for the IRTMs obtained from the compilation of the Antescofo DSL mixed scores. Indeed, the generated models might execute the rule (deadlock) when simulated using the alternative semantics presented in Section 3.2. This problem can be solved with a IRTM transformation procedure which roughly works as follows.

We dissociate the communications between $\mathcal{M}'_{\mathsf{env}}$ and $\mathcal{P}'$, the test environment and the proxy models, and between $\mathcal{P}'$ and the rest of the model $\mathcal{M}'_{\mathsf{sys}}$. We introduce for this purpose a new and fresh signal $se_i \in \mathit{Sig}$ for each $e_i \in \mathit{Evt}$, signaling the detection of the event e_i. We rename the signals

$\overline{e_i}$ into $\overline{se_i}$ because it is an error detection of e_i. After this transformation, all the symbols received in the IRTM, with the exception for $\mathcal{M}'_{\mathsf{env}}$ and $\mathcal{P}'$, are in *Sig*. Moreover, the proxy is modified in order to echo the reception of an event e_{i+1} that causes the emission of an error signal $\overline{e_i}$. The echo has the form of a signal se_{i+1} emitted right after $\overline{e_i}$. This is illustrated in Figure 5.21 for a proxy constructed with $n_{err} = 2$.

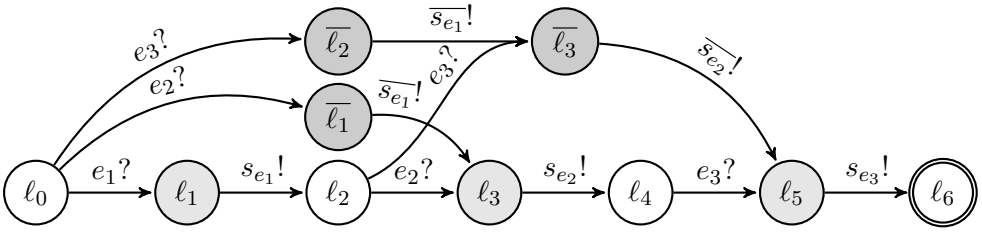

Figure 5.21: Proxy Uppaal TA transformed to prevent from (deadlock).

Execution of Input Trace Set to Antescofo

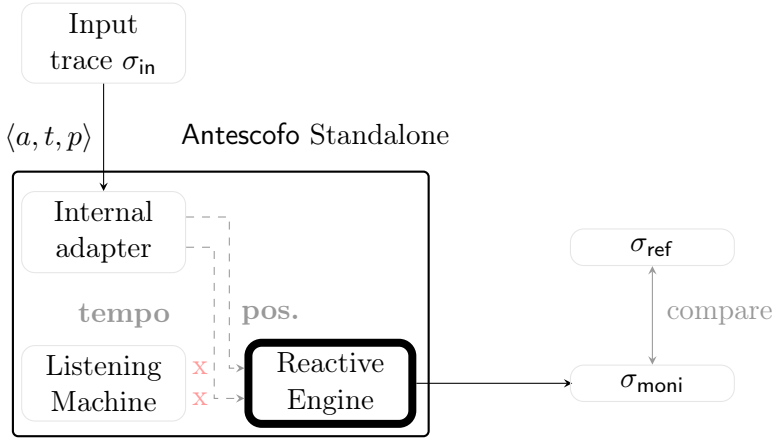

Figure 5.22: Testing the reactive engine.

We implemented several test scenarios for executing the system Antescofo.

1) Reactive Engine Testing. The first test scenario, depicted in Figure 5.22, is performed with a standalone version of Antescofo equipped with

an internal *test adapter* module. The adapter iteratively reads elements $\langle a_i, t_i, p_i \rangle$ of σ_{in} interpreted as a *recognition trace* in a file. The duration $d_i^{\sigma_{\mathsf{in}}} = t_i - t_{i-1}$ of the event e_i, in performance time, is converted into physical time by:

$$d_i^{\mathsf{phy}} = \frac{d_i^{\sigma_{\mathsf{in}}}.60}{p_{i+1}} \tag{5.1}$$

Then, the adapter waits for d_i^{phy} seconds before sending e_{i+1} and p_{i+1} to the RE. More precisely, it does not physically wait, but instead notifies a *virtual clock* in the RE that the time has flown by d_i^{phy} seconds. This way the test does not need to be executed in realtime but can be done in a fast-forward mode. This is very important for batch execution of huge sets of test cases. Notice that we interpret a σ_{in} as a recognition trace, assuming an input sequence from the LM. The messages sent by the RE are logged in σ_{moni}, with timestamps in physical time (*i.e.* with a tempo of 60_{bpm}). In this scenario, the IUT is the RE (the LM is idle).

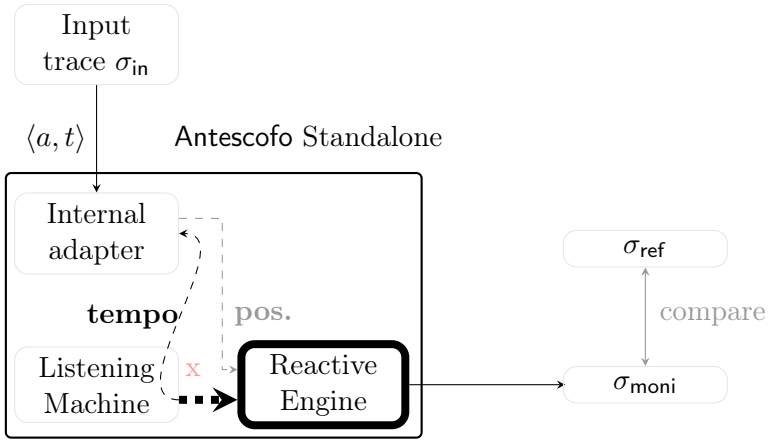

Figure 5.23: Testing the reactive engine and the tempo inference.

2) Reactive Engine and Tempo Inference Testing. In a second test scenario, depicted in Figure 5.23, tempo values p_i read in σ_{in} are ignored by the adapter, which instead uses the tempo values inferred by the LM. The adapter is calling an appropriate method of the LM, in order to compute the events' durations d_i^{phy}. The rest of the scenario is similar to the first scenario. The values of tempo inferred by Antescofo's LM are stored by the adapter and used later to convert the timestamps in the expected output trace σ_{ref} from performance to physical time, in order to be able to compare

it with the real output trace σ_{moni}. In this case, the IUT consists in the RE plus the part of the LM in charge of tempo inference.

Remark that the LM infers the tempo based on the shifts between the durations in σ_{in} and in the mixed score [36]. It might result in a tempo increasing exponentially when durations in σ_{in} are too short.

Example 5.2.30: Let us see how the detected tempo can increase from the ideal trace:

$$\langle e_1, 0, 144\rangle\cdot\langle e_2, \frac{1}{7}, 144\rangle\cdot\langle e_3, \frac{2}{7}, 144\rangle\cdot\langle e_4, \frac{3}{7}, 144\rangle\cdot\langle e_5, \frac{4}{7}, 144\rangle\cdot\langle e_6, \frac{5}{7}, 144\rangle\cdot\langle e_7, \frac{6}{7}, 144\rangle.$$

We execute the trace $\sigma_{\mathsf{in}}^{exp}$ via this scenario:

$$\sigma_{\mathsf{in}}^{exp} = \langle e_1, 0, \_\rangle \cdot \langle e_2, \frac{9}{70}, \_\rangle \cdot \langle e_3, \frac{18}{70}, \_\rangle \cdot \langle e_4, \frac{27}{70}, \_\rangle\cdot$$
$$\langle e_5, \frac{36}{70}, \_\rangle \cdot \langle e_6, \frac{45}{70}, \_\rangle \cdot \langle e_7, \frac{54}{70}, \_\rangle.$$

Each event's durations of this trace is reduced to 10% from the ideal values (lasting $\frac{1}{7}$ beat in the ideal case), because, *e.g.* it is the lowest bound in the model $\mathcal{M}'_{\mathsf{env}}$. The duration of e_1 is computed with the timestamp of e_2 found in $\sigma_{\mathsf{in}}^{exp}$: $d_1^{\sigma_{\mathsf{in}}} = \frac{9}{70} - 0$. Then we obtain a physical value of $d_1^{\mathsf{phy}} = 0.05357$ second with a tempo of 144_{bpm} (the score value by default). The detection of e_2 is earlier than expected and Antescofo's LM modifies its current tempo to 146_{bpm}. The computation of the same relative duration ($\frac{9}{70}$) for the next event is done with a faster tempo and gives $d_2^{\mathsf{phy}} = 0.05283$ second, the event is earlier so the next tempo is faster than 146_{bpm} and so on. In this very short example, we reach at the end a tempo of 150.3_{bpm}, resulted in 6.3_{bpm} more than the score tempo only for 0.4 seconds of performance, it is impossible in practice. ♢

3) Antescofo Testing. We propose a last test scenario, depicted in Figure 5.24. This scenario is the most general and is executed with a version of Antescofo embedded in MAX-MSP (as a MAX-MSP patch), using an adapter which is another MAX-MSP patch. The adapter iteratively reads triples $\langle a_i, t_i, p_i\rangle$ in a file containing the trace σ_{in}, and converts them into MIDI events, with durations d_i^{phy} casted into physical time using (5.1). The events are played by the MAX-MSP patch midisynth~ and the audio stream generated is sent to the LM. The output of the RE is then traced in σ_{moni} as before.

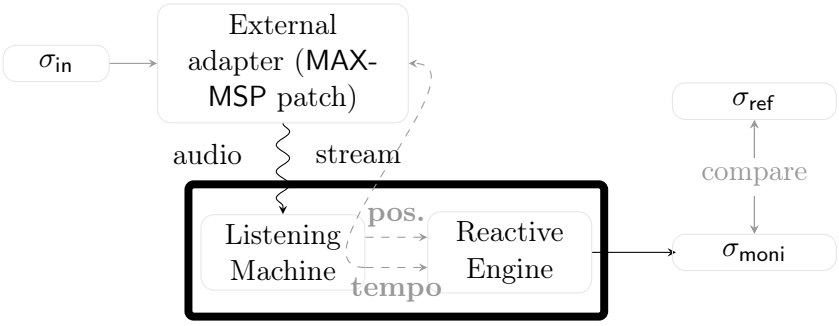

Figure 5.24: Testing the whole Antescofo system.

Note that here, the RE uses the tempo values detected by the LM, which may differ from the tempo values in σ_{in}. The detected tempo values are saved by the adapter, indeed, in MAX-MSP the detected tempo is available as an outlet of the antescofo~ patch. They are used later to convert the dates in σ_{ref} from musical into physical time, like in the previous scenario. In this realistic scenario, the IUT is therefore the whole Antescofo system.

In an alternative scenario, the adapter uses the tempo values p_i in σ_{in} for computing the events' durations d_i^{phy}, like in the second scenario.

Note that in both the last two scenarios, the tests are executed in real-time and not in a fast-forward mode. However an audio file of the sequence of MIDI sounds can be recorded and sent later to the standalone in fast-forward mode.

Online Testing Approach

The *online* approach uses the VM directly on the IRTM, hence the translation restrictions do not hold here. The *loops* actions and multi-time durations can then be kept in the model.

Figure 5.25 illustrates the *online* approach, the implementation follows the VM description in Section 3.3 and its generation algorithm in Section 4.3.1. The execution is done with the standalone version of Antescofo and the internal adapter detailed into the scenario 1. However, the adaptor uses OSC reception for reading from a file the events and amounts of time to advance.

This adapter allows us to use the fast-forward feature even for the real-time testing, increasing the possibilities of the *online* approach.

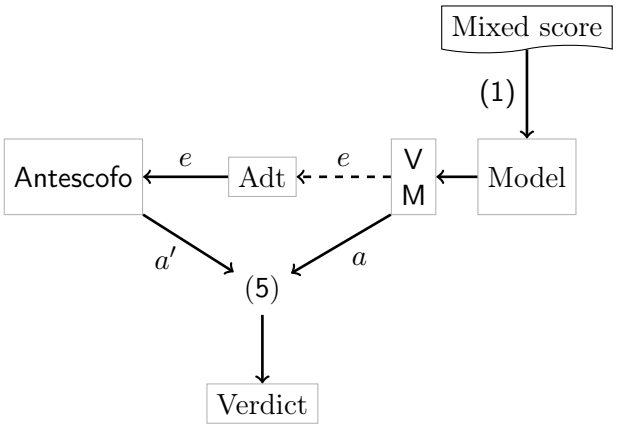

Figure 5.25: Online Score-based IMS testing procedures.

Verdict

The offline and online approaches output a verdict, detailing the behavior of Antescofo according to its reference for each input trace of the tests.

The verdicts are produced offline by a tool comparing the expected and monitored traces σ_{ref} and σ_{moni} with an acceptable latency ε, generally about 0.1 ms. A verdict is pretty printed to inform the testers on the conformance of Antescofo to the models. We mark as errors unexpected or missed atomic actions sent or not by the IUT and a delay more than ε ms between the model and the system actions. The document is split in logical instants in order to visualize clearly the sequence of actions related to an external event reception. The verdicts also detail the variations between the input trace and the ideal trace in order to outline early or late events, which are not always easy to detect.

Example 5.2.31: We depict in Figure 5.26 a verdict with the example presented in Section 5.2.2 and for the attributes tight, global (meaning that the timed-requirements modeled here are those specified in the mixed score). The input trace is the ideal one with an event end to stop the test explicitly: $\langle e_1, 0, 120\rangle \cdot \langle e_2, 1, 120\rangle \cdot \langle \mathsf{end}, 2, 120\rangle$. The verdict follows the monitored trace (on the left) and details the reference values (on the right), depicting: the label, the physical timestamp, called *now* in Antescofo, and the relative score timestamp, called *rnow*, for each item. Notice that the labels on the right change to be compatible with Uppaal model checker, and that the score tempo is initially set to 120_{bpm}. Each time advancement is compared to detect any

```
-------------------------------------------------------------------------------
|-        Antescofo Trace                |          Expected Trace          -|
|-    label      now      [rnow]         |label  comp. timestamp [ref beat] -|
|-----------------------------------------------------------------------------|
|         a1       0        [      0]   |         a0       0       [      0]|
|         e1       0        [      0]   |         e0       0       [      0]|*     120BPM
 +   0.25 (    0.5 *     120)           ==   0.25 (    0.5 *     120)
|         a2    0.25        [    0.5]   |         a1    0.25       [    0.5]|
 +   0.25 (    0.5 *     120)           ==   0.25 (    0.5 *     120)
|         a3     0.5        [      1]   |         a2     0.5       [      1]|
|         b1     0.5        [      1]   |         a4     0.5       [      1]|
|         e2     0.5        [      1]   |         e1     0.5       [      1]|*     120BPM
 +   0.25 (    0.5 *     120)           ==   0.25 (    0.5 *     120)
|         a4    0.75        [    1.5]   |         a3    0.75       [    1.5]|
|         b2    0.75        [    1.5]   |         a5    0.75       [    1.5]|
 +   0.25 (    0.5 *     120)           ==   0.25 (    0.5 *     120)
|        END       1        [      2]   |         e2       1       [      2]|*     120BPM
|-----------------------------------------------------------------------------|
 Checked :: Test OK
```

Figure 5.26: A verdict returning pass.

time differences. Moreover, the interpretation information is depicted with the ideal trace. The returned verdict is pass, denoted by OK, and assesses the IMS conformance for this input trace. ♢

5.3 Experiments

Our TMBT framework allows us to assess the IMS Antescofo conformance to a mixed score. However, we want to evaluate the effectiveness of our TMBT framework and report the pros and cons of the different approaches.

Here, we want to measure a black-box testing framework and thus assume that we have no feedback in the coverage of the Implementation Under Test's line codes and specially that we do not know the erroneous lines. We then take as a metric of effectiveness:

- the coverage, in the IRTM locations, of the suite of test cases generated, and
- the size of the input score, to evaluate state explosion for real cases.

The first criteria ensures the scalability of our framework, implying the possibility to test real mixed-scores. The second criteria allows us to judge the

quality of the tests, indeed, greater is the coverage better the rtioco conformance [44, 56] should assure a good time reliability for the system under test.

This section evaluate the framework, following three algorithms for generating input traces, presented in Section 4.3. First, we test the CoVer generation method and the fuzzing generation to compare their pros and cons *wrt.* the scalability and coverage measures. Then, we present our early online testing framework and compare its outcome to the offline results.

The results were obtained with a MacBook Pro Retina with a 2.3 GHz Intel Core i7 and 16Go 1600 MHz DDR3 of Memory. The laptop ran on the El Capitan version of MAC OS X (10.11.6).

Case studies Presentation

We have considered two case studies in our experiments:

1. a benchmark made of hundreds of little mixed scores, covering many features of the IUT's DSL

2. a real mixed score of the piece of *Sonata in F major, HWV 369 third movement: Alla Siciliana* by *Georg Friedrich Händel* [4].

The first benchmark is useful to provide tests for the development (debugging and regression tests) of the system Antescofo. It aims at covering the functionality of the system's DSL and checking the reactions of the IMS. The second mixed score is a long real test case, to evaluate the scalability of our test methods. Its total size is 1018 events and 3237 actions gathered in one big group in order to do automatic accompaniment by sending MIDI notes. This second case study is split into five extracts: the first 5^{th} bars (25 events and 84 actions), 8^{th} bars (48-185), 10^{th} bars (74-264), 15^{th} (122-444) and 40^{th} bars (360-1218).

Each case study is processed with various values for n_{err} and κ, the numbers of possible consecutive errors and the bound on the variation of event's durations. For the results, we used the VM developed for online testing in order to compute the coverage of the suite of test cases generated for each experiment.

[4] You can have a quick representation of the piece (with a description (in French) of Antescofo) here:
https://interstices.info/jcms/c_17524/interaction-musicale-en-temps-reel-entre-musiciens-et-ordinateur

5.3.1 Results with Offline Approach

Covering generation

We evaluated the generation of test data with Uppaal/CoVer following the offline method presented in Section 5.2.3. The script creates the IRTMs, translates them into networks of TA, generates test suites using CoVer, executes them according to *the first* scenario presented Section 5.2.3 and compares the outcome to test cases.

κ (%) \ n_{err}	0	1	3	5	7
0	18,231 - 051	19,402 - 107	23,377 - 164	26,424 - 313	26,443 - 307
1	18,231 - 025	19,402 - 076	23,377 - 137	26,424 - 319	26,443 - 318
3	18,111 - 024	19,402 - 077	23,377 - 137	26,424 - 318	26,443 - 314
5	18,231 - 028	19,402 - 077	23,377 - 132	26,424 - 312	26,443 - 309
10	18,231 - 035	19,402 - 082	23,377 - 140	26,424 - 320	26,443 - 323
25	18,231 - 059	19,402 - 086	23,377 - 135	26,424 - 334	26,443 - 328
50	18,231 - 103	19,402 - 132	23,377 - 160	26,424 - 354	26,443 - 352

Table 5.1: CoVer on the benchmark: the total size in number of IRTM states (on the left) - the time in seconds to perform the whole script (on the right).

κ (%) \ n_{err}	0	1	3	5	7
0	754 - 67.78%	1674 - 82.33%	2781 - 87.95%	3271 - 87.60%	3262 - 87.28%
1	612 - 67.79%	1471 - 81.56%	2615 - 87.75%	3183 - 88.16%	3152 - 87.44%
3	609 - 67.75%	1456 - 80.42%	2635 - 87.94%	3160 - 87.05%	3162 - 87.41%
5	613 - 67.97%	1468 - 81.54%	2633 - 87.81%	3140 - 87.07%	3124 - 86.06%
10	715 - 68.01%	1513 - 81.51%	2681 - 87.90%	3191 - 87.37%	3201 - 87.56%
25	994 - 68.18%	1720 - 82.36%	2691 - 87.60%	3243 - 88.29%	3277 - 88.02%
50	1623 - 69.22%	2301 - 83.41%	3006 - 88.82%	3577 - 88.34%	3553 - 88.54%

Table 5.2: CoVer on the benchmark: the number of σ_{in} generated (on the left) - their related coverage (on the right).

Tables 5.1 and 5.2 report the results with different environment options for all the scores in the benchmark. The first table details the total size of the model part $\mathcal{M}_{\mathsf{sys}}$ in number of IRTM locations and the total time to execute the whole benchmark. The second table presents the number of input traces generated by CoVer and the coverage on $\mathcal{M}_{\mathsf{sys}}$ locations of their related test cases.

The same script was ran for the extracts of the real mixed-score and the results are reported in Table 5.3. The table depicts the number of input traces generated with their total coverage for each extract, denoted by its number of bars. The size of the IRTM $\mathcal{M}_{\mathsf{sys}}$ is 328, 697, 1005, 1678 and

n_{err}-κ (%) \ bars	5	8	10	15	40
0-00	1 - 43.59%	1 - 38.92%	1 - 39.01%	1 - 38.72%	1 - 38.72%
0-10	38 - 43.59%	74 - 38.92%	117 - 39.01%	262 - 38.72%	×
0-25	95 - 43.59%	201 - 38.92%	427 - 39.01%	×	×
3-00	84 - 66.66%	130 - 86.48%	×	×	×
3-10	85 - 66.66%	148 - 86.48%	×	×	×
3-25	94 - 66.66%	159 - 86.48%	×	×	×
7-00	113 - 96.94%	×	×	×	×
7-10	133 - 96.94%	×	×	×	×
7-25	147 - 96.94%	×	×	×	×

Table 5.3: CoVer on the real case: number of σ_{in} generated (on the left) - their related coverage (on the right).

4668 locations for the extracts of 5, 8, 10, 15 and 40 bars of the mixed-score respectively. In Table 5.3, the crosses depict a state explosion during the generation of input traces, because no output was returned or because a crash happened during one of the script steps.

Feedback. The advantages of the CoVer generation are its effectiveness to generate covering test suites for the first case study. This case study contains a lot of small-sized mixed-scores that is perfect in such a case. Moreover, the amount of time is correct since the scripts spent 352 seconds to generate and test 3553 input traces, an average of 10 seconds per input trace, with a good coverage on 88.5% of the model locations. However, the inconvenient are also multiples. We have not a clear control on the coverage according to the environment parameters. For example, the possibility of missing one more event improves more the coverage than allowing more interpretation on the durations. The real case shows clearly the lack of scalability because the extracts of more than 10 bars (74 events and 264 actions) cannot be tested with errors.

The CoVer generation is efficient for toy-examples where the mixed scores are written in a purpose of debugging. However, CoVer cannot be satisfying for real cases. The input traces are commonly generated with the lower values in their durations, because of the guards in the model. These values are not musically relevant since when converting performance time into physical time, having an input trace with shortest delays may result in a geometric progression of the tempo inferred by Antescofo, leading to exponential accelerations and unrealistic tempo values (for example a tempo of 300_{bpm}) as presented in Section 5.2.3 for the second scenario of simulation. These weaknesses encouraged us to explore other approaches for test data generation

and execution as detailed in Section 4.3 which can be used to circumvent this problem.

Fuzzing generation

For the fuzzing generation, the script first creates the IRTMs without environment model $\mathcal{M}_{\mathsf{env}}$, then, fuzzes the ideal trace to create a set of input traces and translates the IRTMs, with a specific $\mathcal{M}_{\mathsf{env}}^{\sigma'_{\mathsf{in}}}$ for each input trace, into networks of TA. With these models, the script simulates the TA using Verifyta, the Uppaal model checker command tool, to compute the reference traces. Then, the script executes the input traces according to the first scenario presented in Section 5.2.3 and compares finally the two output traces.

bars / n_{err}-κ (%)	5	8	10	15	40
0-00	1 - 38.81%	1 - 14.67%	1 - 38.27%	1 - 38.05%	1 - 38.02%
0-10	10 - 38.81%	10 - 14.67%	10 - 38.27%	10 - 38.05%	10 - 38.04%
0-25	10 - 38.81%	10 - 14.72%	10 - 38.27%	10 - 38.11%	10 - 38.02%
3-00	10 - 37.65%	10 - 21.15%	10 - 34.21%	10 - 28.72%	10 - 23.21%
3-10	10 - 41.91%	10 - 34.69%	10 - 32.75%	10 - 32.59%	10 - 27.91%
3-25	10 - 28.05%	10 - 28.83%	10 - 28.48%	10 - 27.85%	10 - 25.89%
7-00	10 - 17.03%	10 - 17.21%	10 - 17.26%	10 - 16.37%	10 - 15.60%
7-10	10 - 17.68%	10 - 17.36%	10 - 16.76%	10 - 16.21%	10 - 15.90%
7-25	10 - 18.01%	10 - 18.09%	10 - 16.55%	10 - 16.53%	10 - 15.66%

Table 5.4: Fuzz on the real case: number of σ_{in} generated - coverage according to each extract and the different environment restrictions ($n_{err} - \kappa$).

The values are depicted in Table 5.4 for the real mixed-score and with the fuzz generation. We do not report the amount of time to process the tests since we cannot compare a script doing an input generation against a random fuzzing one. To have an idea, the extract of 5 bars with parameters 7-25 lasted 97 seconds for a test of one input trace using Verifyta.

The advantage of the fuzzing generation is the little deformations of the ideal trace, that keeps the input traces musically relevant. Moreover, in a musical point of view, we think that a little interpretation is sufficient to consider later, earlier or missed cases, *i.e.* have a good coverage on the musician performances. The method is fast and can manage huge mixed-scores that is good for real cases. However, since the fuzz is done randomly we have no control on the coverage which is low for a set of input traces.

number of σ_{in}	1	10	100	1000
percentage of coverage	12.72%	15.78%	21.95%	32.35%

Table 5.5: Fuzz: number of σ_{in} generated for 40 bars of the real mixed score with parameters 7-25.

Evaluation of the coverage of fuzzing generation

Random test is an important strategy for test input generations and is widely used in the state-of-the-art. However, it lacks of precision since no control is possible in the randomized values. We add an experiment to evaluate the coverdness of our fuzzing script according to the number of input traces generated with 7-25 values for the parameters n_{err} and κ respectively.

We present in Table 5.5 the results on the first 40^{th} bars of the mixed score of the Sonate used as the second case study (see page 145). The rise of the generated trace number improves as expected the coverage, but it is still very low even for a thousand of traces, that lasts as long as a CoVer generation. This experiment confirms that this second generation cannot be covering and motivates us for another strategy or targets addition for guiding the fuzz algorithm.

5.3.2 Results with Online Approach

Finally, we report in Table 5.6 an evaluation of our online testing approach. For the online experiment, we deployed a script running the Virtual Machine (VM) and the IUT Antescofo at the same time on the machine, the two softwares communicate via the protocol Open Sound Control (OSC). The VM constructs and simulates the model following the method and the algorithm detailed Section 4.3.1. The IUT is run with an online adaptor which waits for an input stimulation, an event symbol or a duration, from the model. Recall that, although the method is online, we execute it in a fast-forward fashion, preventing from waiting for the real durations.

The online framework is promising. Our first experiments succeeded in

number of σ_{in}	10	50	100
percentage of coverage	59.32%	62.09%	62.09%
time in seconds	24	114	249

Table 5.6: Online: number of σ_{in} generated for the all mixed score (18.641 model's states).

managing the entire real mixed score (case study 2) and performed a hundred of input traces for 4 minutes. We are working on improving the online algorithm *wrt.* coverdness of the test suite generation. In particular, we are working on improving the distribution of input events, using constraint solving techniques like in SAGE [20, 51].

Summary

In this chapter, we applied our TMBT framework to a state-of-the-art IMS. We first presented the IUT of our application, the score-based IMS Antescofo. Then, we detailed the abstract syntax handled by the construction rules and their definition for constructing IRTMs from every mixed-score included in the syntax. We detailed the possible implementation and parameters provided by the framework to test Antescofo. Finally, we reported experiments of the offline and online approaches with the possible generation algorithms implemented in our framework. The results allow to compare the different options and present the application of our testing procedure to Antescofo on real cases.

Because it is a young framework and is applied to a constantly improving system, the accuracy of raised errors is sometimes difficult to assess. However the errors raised by the framework are timed errors since they come from a wrong output or a wrong timing of an output. Actually we reported errors which came from a concurrency problem which disturbed the scheduling of the outputs, a communication/synchronization problem regarding the inputs/outputs, a wrong time computation (from the tempo updating function) and a wrong management of the specified group's attributes. Moreover several of these errors happened for non trivial input cases, making them hard to find by other means.

These abilities to find errors in Antescofo confirms the well utility of our test framework and encourages us to extend its applicability to IMS in general.

Chapter 6

Conclusion and Perspectives

In this thesis, we presented a fully automatic Timed Model-Based Testing (TMBT) framework dedicated to real-time Interactive Music Systems (IMS). One originality of the case study is that the models are constructed automatically from the mixed scores required by the score-based IMS, instead of being written manually by an expert.

First, we presented the model of our framework, called Interactive Real-Time Model (IRTM), specifying the implementation under test and its environment (*i.e.* the human musicians accompanied). IRTMs are designed to model easily the semantics of a score-based IMS, and in general, both time-triggered and event-driven systems. Moreover, in order to specify musical scenarios, the model provides durations related to different time units.

IRTMs borrow both from the Timed Automata model and the logic time semantics of the synchronous programming languages for reactive systems Esterel [13]. In order to simulate such models, IRTMs can be interpreted as a byte-code by a virtual machine or can be translated (under restrictions) into Timed Automata for using tools of the Uppaal suite.

Then, we exposed our TMBT framework, based on IRTMs. This framework permits to assess the IMS timed conformance to a mixed score and provides multiple options such as the offline and online approaches for the generation of covering and relevant test cases.

Finally, a case study was presented and consist in the application of our framework to Antescofo. Antescofo is a constantly evolving score-based IMS

used in live on current music productions. We presented the implementations of our framework for the IMS and reported the results of the experiments in order to evaluate the framework efficiency.

To conclude we answered to the question: "Can formal methods be used to test embedded systems ?" with encouraging perspectives. Effectively, we implemented a framework to test an IMS and applied it to real test cases via concrete experiments.

During these testing procedures, one technical difficulty was the necessity to deal with different time units, in particular the musical time relative to a tempo. This problem prevented us from using the online testing tool Uppaal Tron out of the box for our case study. Hence, we implemented our own online MBT framework using a virtual machine interpreting directly the IRTM without any restrictions. This newly method is yet promising since an entire real mixed score passed successfully a first experiment using a non trivial "on-the-fly" algorithm for the generation of test input data.

Our method is designed to test the behavior of the IMS on one given score, by generating a covering set of input traces describing a range of musical performance of the score. This approach is advantageous both for IMS debugging, thanks to coverage criteria, and for user assistance to author mixed scores, using the fuzz generation based on models of musical performance. The framework detected successfully some errors from the implementation under test and is used to store a number of tests for regression purpose. We argue that it is yet a good framework for testing real-time systems.

Besides the case of IMS, our approach could be applied to the test of other real-time reactive systems involving pre-specified temporal scenarios, feedback and timed interaction with humans. It is motivated by the formalization we provided describing the IRTM syntax and semantics with its related test framework. These descriptions manage abstract symbols that can be mapped to several applications. Briefly, we can apply in general the framework to test cyber-physical systems coupling computing devices with physical components and humans in the loop.

We terminate this manuscript with a discussion, the related work and the possible perspectives of our work.

6.1 Discussions

In this section, we discuss about the characteristics of the test framework and expose the related work. The discussion will follow the workflow of our MBT framework, exposing our remarks for each step. Then, we highlight in the related work, how they are linked to the existing approaches explaining the differences and the choices we have made in the application of our TMBT framework to our peculiar context.

Construction. The automatic model construction brings an easy way to make models. It requires an abstract syntax and related construction rules in order to automatically build a model from whatever requirements included in the abstract syntax.

In the case of Antescofo mixed score, the complexity of its model construction depends on the number of event N_e and actions N_a in the mixed score. The worst case consists in having N_a groups requiring a traversal of the entire score, giving a complexity of $\mathcal{O}((2 + N_a) * N_e) + N_a)$ in time (the 2 comes from the proxy $\mathcal{P}$ and the environment model $\mathcal{M}_{\mathsf{env}}$), with $\mathcal{O}((8 * N_a) + 2n_{err} * N_e)$ locations and $\mathcal{O}((8 * N_a) + 2n_{err} * N_e)$ transitions in space (supposing the group matching the worst tight global cases). Remark that the real number of actions for this case is N_a^2 because it is required to have at least one atomic action in a group. In practice, we have a quadratic complexity in time and a linear complexity in space, notice moreover that the construction algorithm traverses the AST using the visitor pattern that is pretty fast in time.

Then, during the translation, we require 3 additional model traversals that has no impact on the entire computation time but adds 4 internal signals per group, 2 internal signals per event and $((N_e * N_a) + 4N_a)$ transitions on the resulted TA. This explosion is due to the need in an Uppaal TA model (at least when using CoVer) to explicitly specify the environment action receptions. Unfortunately, to prevent from losing a possible test case, we need to add for every suspending location (*i.e.* for every event) one reception for every action in the model.

The modular approach provided by the connectors (the providers, seekers and their operators) makes the model construction compositional. In the sense that add a group or compose models or sub-models becomes easy. This modularity assures the scalability of the compilation process of the model.

The construction procedure is not totally without effort since the rules need to be written by testers. However, they construct little parts of the model according to the abstract syntax specifying the system. Moreover,

they allow to automatically build for any requirements following an abstract syntax their related model. The automatic construction greatly eases and clarifies the specification by linking to the syntax a part of model composed during construction. Once specified, the test procedure becomes effectively fully automatic, from the requirement to the verdict.

Generation. We have two approaches to implement test data generation methods, the former, offline, ensures a good quality of the test data via covering features; the latter, online, scales to big scores and fits the musical context of the generated traces. The main drawbacks of these approaches are the scaling problem of the former method and the coverage management of the latter. However, the fuzzing generation permits to process real cases and manages TIFs functions to guide the tests. This last method can be improved and adapted to the *online* algorithm for solving its weakness of coverage.

Remark that implementing alternative generation methods was highly motivated by the main limitations of the CoVer offline approach. Specially because the input traces are not musically relevant (because of CoVer which strictly follows the model constraints). However, this approach is well suited for debugging the system Antescofo, using small ad-hoc scores (see Section 5.3).

Execution. The blackbox decision during the execution step is not an usual case in MBT frameworks. The possibility to target the whole system or a discrete module provides a new vision of testing. Thanks to the multi-time input traces, different interpretations allow to test different IUT's modules with the same model. In our case, we can deduce the variations on the output of the system with or without the LM module for the same input trace, *i.e.* testing only the RE or the whole system. It may be another way to check the reliability and the relevance of complex physical-recognition/following algorithms by composing the module tested themselves onto the blackbox.

6.2 Related Work

Some tools exist for automating the test of IMS, like for instance the MAX-test package [73] for testing MAX patches through assertions. These systems conveniently provide sophisticated tools for automating execution of test data and reporting, but they generally do not offer procedures for generating test data. Hence, the user must compute some input test data and the

expected corresponding output by other means. Our approach in contrast focus on the generation of test data, based on formal models, and in this respect the two approaches can be seen as complementary.

Other works have addressed the formal verification of multimedia systems based on TA models, as for instance the verification of a lip-synchronisation protocol (synchronization of audio and video streams) in [24]. Model checking procedures have also been used for music improvisation [11]. TA, Uppaal, as well as timed Petri nets, are used in i-Score [8], a framework for composition, verification and real-time performance of Multimedia Interactive Scenarios. To our knowledge, no other work has applied such formal models to the test of IMS.

One drawback of many MBT methods is that they generally require a manual expert intervention for the construction of models. Some specification-based testing procedures also involve the automatic construction of models from high-level user specification of test case scenarios. For instance, in [34] the specifications are written in the quasi-natural language Gherkin and the models are used with the model-based testing tool QuickCheck. In our case, the mixed scores can be considered as a complete specification of all the possible timed scenarios rather than some test case scenarios. Moreover, our specifications are not written for test purpose but prior to the execution of the system during an interactive performance. Hence, they are defined by users before the time of testing and no more intervention is needed during the test workflow. Therefore, our test procedure involves temporal values that is not common in MBT with an automated construction of timed models.

The system GUITAR, presented in [70], proposes to use static analysis and semi-automatically reverse-engineer methods for constructing an event-flow model from an implementation of the GUI functions of JAVA programs. Notice that we are producing our IRTM models by analysis of mixed scores too, however, we are not testing the written mixed score but the interpreter (*i.e.* the system) executing this score. More precisely, a IRTM model is produced by parsing a mixed score and traversing its abstract syntax tree, using sequential and concurrent composition operators for the IRTM, similar to the glue operators in [15, 16]. This approach is modular in the sense that the model of several scores can be combined into a larger model using these operators.

Moreover, our IRTMs are also executable, similarly to the Ecode of [54, 50], which is obtained by compilation of programs in the time-triggered programming Giotto language and is used for static analysis of properties such as time-safety or schedulability. An earlier version of the IRTM presented in this paper has also been used for analysis of the robustness of

Antescofo mixed scores [47].

Our approach for the offline generation of test input by fuzzing ideal performances is inspired by fuzz testing. In [21] the fuzzing method is used in a white box fashion in a large scale testing framework. Although we follow a black-box testing approach, we use the same strategy which consist in starting from a perfect input and mutating it. Note that, in contrast to most fuzz testing approaches, we needed to deal with time values in this context, and applied for this purpose models of music performance from the literature.

6.3 Future Work

Several possible perspectives could improve our testing framework or explore its features. We present some of them in this section.

Specification and Model Construction. The model construction is based on some requirements written by the user. In case of scored-based IMS, the requirement file is the score, which is mandatory for the system to run. Nevertheless, in general one may think that we "delayed" the specification problem to the users side. Indeed, the testers just need to develop a concrete language to fit the abstract syntax parsed by the construction rules. A future work can be to generalize such syntax, here implemented by the Antescofo's DSL, to provide a language for expressing easily common specifications of real-time systems. We have already mentioned, in the related work Section 6.2, the language Gherkin[1] which uses the simple *Given-When-Then* format to specify system scenarios with a quasi-natural language. In a general case, we can imagine the specification of an ideal scenario in which a list of events describes the actions they trigger, and how these events are timely linked together.

Offline Test Suite Generation. The offline test generation approach based on CoVer is a good first step into applying some existing MBT tools to our case study, with a purpose of exhaustiveness. In our case however, it has some limitations and cannot be considered as the best way to generate input traces. Indeed, this approach builds a trace σ_{in} with duration values restricted to the shortest delays that causes exponential accelerations, it is not a scalable approach and needs to hold the translation restrictions. A way to bypass these problems could be to re-implement (in Uppaal) the

[1] Available at: `http://github.com/cucumber/cucumber/wiki/Gherkin`.

algorithm of CoVer with random choices of delays insides regions, instead of the systematic choice of the shortest delays. Indeed, the Alur and Dill's region graph [2] partitions a TA model in terms of future behaviors, the states which can fire the same set of discrete transitions are merged. These graphs can be used to easily compute our coverage during the generation, fuzzing a little our input timing variables to change the regions reached by the set of generated traces.

Online Testing and Virtual Machine. The VM can compute the coverage of the IRTM model $\mathcal{M}_{\mathsf{sys}}$ for a given set of σ_{in}. This information could be very useful in order to assess the quality of a set of input traces independently generated (*e.g.* using a set of TIFs). Formal models can be developed and used in the online generation to compute a covering set of input traces.

A work has been started adding probability values in the model. The technique must be implemented in the test framework as a possible option of input traces generation and experiments must be performed to compute the efficiency of this method. In [9], a uniform sampling for TA is proposed to tackle the problem of language inclusion measurement. This method can be adapted for our generation purpose by computing from a TA (or the equivalent IRTM) a relevant sampling of probability values, to explore the model.

However, along these algorithms, the main question summarizing these generations of input traces for IMS is: *Can we highlight a musical sense to a model coverage ?* And *Is an input trace with a good coverage sufficient to consider all the possible performances we can confront to a IMS?*

Visualization. In Section 5.1, where Antescofo is presented, we briefly introduced its open-source visualization software: Ascograph. A recent work has been performed to improve this software and add more features. A possible improvement would be to add a monitored trace view to this software, and run Antescofo with an input trace created with Ascograph. As a result, a *debugger* view can be added, where an input trace can be built by the user, resulting in a display of the related reactions emitted by the system. The debugger can provide more complex and interesting feedback on the score from the test framework. The IRTM graphical view can play a role in the understanding of the system reactions and can be used for debugging.

Time-Safety. During performance of Antescofo, MAX-MSP manages the time scheduling and the sound processing of the system. An Antescofo's

improvement consists in be free from the environment of MAX-MSP, managing the real-time clock computations and audio/control balance by itself. It is a well-known problem for mixed-music and IMS softwares and is already a study presented in [45]. The complexity consists in allocating time for sound-processing and generation, which requires a non negligible time resource consumption. Meanwhile, the variable computations for the control requirement of the software must be performed.

In order to tackle the problem, the IRTM can be extended with ports and task features separating the complex and non instantaneous functions from the fast IRTM code. It will allow time-safety analyses and a scheduling computation in order to assure the possible execution of a piece and to create watchdogs in case of dynamical changes during a performance. This method is inspired by the Logical Execution Time [62] (LET) abstraction, which combines the Bounded and Zero Execution Time (resp. BET and ZET) in a byte-code assuring time-safety of multi-task systems (using their WCET value). It was concretized with the time- and event-time-triggered language Giotto [54] and XGiotto [50], presented in the related work, successfully applied to a realtime drone system. Moreover, such syntax will permit to specify more features of the Antescofo DSL and test the whole system.

Bibliography

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, May 1993.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[3] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–203, January 1994.

[4] David P. Anderson. A system for computer music performance. Technical report, Berkeley, CA, USA, 1989.

[5] David P. Anderson and Ron Kuivila. A system for computer music performance. *ACM Transactions on Computer Systems (TOCS)*, 8(1):56–82, February 1990.

[6] Jaime Arias. *Formal Semantics and Automatic Verification of Hierarchical Multimedia Scenarios with Interactive Choices*. Phd, Université de Bordeaux, France, 2015. PhD Thesis.

[7] Jaime Arias, Myriam Desainte-Catherine, and Camilo Rueda. A Framework for Composition, Verification and Real-Time Performance of Multimedia Interactive Scenarios. In *15th International Conference on Application of Concurrency to System Design*, Piscataway, USA, June 2015. IEEE.

[8] Jaime Arias, Myriam Desainte-Catherine, and Camilo Rueda. A Framework for Composition, Verification and Real-Time Performance of Multimedia Interactive Scenarios. In *15th International Conference on Ap-*

plication of Concurrency to System Design, Brussels, Belgium, June 2015.

[9] Benoît Barbot, Nicolas Basset, Marc Beunardeau, and Marta Kwiatkowska. Uniform sampling for timed automata with application to language inclusion measurement. In *Quantitative Evaluation of Systems - 13th International Conference, QEST 2016, Quebec City, QC, Canada, August 23-25, 2016, Proceedings*, pages 175–190, 2016.

[10] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[11] T. Bazin and S. Dubnov. Constrained music generation using model-checking. In *Proceedings of Journées d'Informatique Musicale*, 2016.

[12] A. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In *Model-Based Testing of Reactive Systems*, pages 391–438. Berlin ; New York : Springer, 2004.

[13] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[14] Simon Bliudze and Joseph Sifakis. A notion of glue expressiveness for component-based systems. In *Proceedings of the 19th International Conference on Concurrency Theory*, CONCUR '08, pages 508–522, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Simon Bliudze and Joseph Sifakis. A notion of glue expressiveness for component-based systems. In *Proceedings of the 19th International Conference on Concurrency Theory*, CONCUR '08, pages 508–522, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] Simon Bliudze and Joseph Sifakis. *Software Composition: 10th International Conference, SC 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, chapter Synthesizing Glue Operators from Glue Constraints for the Construction of Component-Based Systems, pages 51–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[17] Simon Bliudze and Joseph Sifakis. Synthesizing glue operators from glue constraints for the construction of component-based systems. In *Proceedings of the 10th International Conference on Software Composition*, SC'11, pages 51–67, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and generating test cases using observer automata. In *Proceedings of the 4th International Conference on Formal Approaches to Software Testing*, FATES'04, pages 125–139, Berlin, Heidelberg, 2005. Springer-Verlag.

[19] Bert Bongers. Exploring novel ways of interaction in musical performance. In *Proceedings of the 3rd Conference on Creativity & Cognition*, C&C '99, pages 76–81, New York, NY, USA, 1999. ACM.

[20] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. Technical Report MSR-TR-2012-55, Microsoft Research, 2012.

[21] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. Technical Report MSR-TR-2012-55, Microsoft Research, 2012.

[22] Patricia Bouyer. Model-checking timed temporal logics. *Electronic Notes in Theoretical Computer Science*, 231:323 – 341, 2009. Proceedings of the 5th Workshop on Methods for Modalities (M4M5 2007).

[23] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of {TPTL} and {MTL}. *Information and Computation*, 208(2):97 – 116, 2010.

[24] H. Bowman, G. Faconti, J-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip synchronisation algorithm using uppaal. In *Proc. of the 3rd International Workshop on Formal Methods for Industrial Critical Systems*, pages 97–124, 1998.

[25] Howard Bowman, Giorgio Faconti, and M. Massink. Specification and verification of media constraints using uppaal. In *5th Eurographics Workshop on the Design, Specification and Verification of Interactive Systems, DSV-IS 98*, Eurographics Series, pages 261–277. Springer-Verlag, August 1998.

[26] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, pages 546–550, London, UK, UK, 1998. Springer-Verlag.

[27] Jean Bresson, Carlos Agon, and Gérard Assayag. Openmusic: Visual programming environment for music composition, analysis and research. In *Proceedings of the 19th ACM International Conference on Multimedia*, MM '11, pages 743–746, New York, NY, USA, 2011. ACM.

[28] Jean Bresson and Jean-Louis Giavitto. A Reactive Extension of the OpenMusic Visual Programming Language. *Journal of Visual Languages and Computing*, 25(4):363–375, 2014.

[29] Ed Brinksma, Laura Brandán Briones, and Mariëlle Stoelinga. A semantic framework for test coverage. In S. Graf and W. Zhang, editors, *Proceedings of the fourth international symposium on Automated Technology for Verification and Analysis (ATVA'06)*, LNCS. Springer, 2006.

[30] Grigore Burloiu and Arshia Cont. Visualizing Timed, Hierarchical Code Structures in AscoGraph. In *International Conference on Information Visualisation*, Barcelona, Spain, July 2015. University of Barcelona.

[31] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 178–188, 1987.

[32] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996.

[33] Thomas Coffy, Jean-Louis Giavitto, and Arshia Cont. AscoGraph: A User Interface for Sequencing and Score Following for Interactive Music. In *ICMC 2014 - 40th International Computer Music Conference*, pages 600–604, Athens, Greece, September 2014.

[34] Christian Colombo, Mark Micallef, and Mark Scerri. Verifying web applications: From business level specifications to automated model-based testing. In *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014.*, pages 14–28, 2014.

[35] Arshia Cont. A Coupled Duration-Focused Architecture for Real-Time Music-to-Score Alignment. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32:974–987, June 2010.

[36] Arshia Cont. A coupled duration-focused architecture for realtime music to score alignment. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 32(6):974–987, 2010.

[37] Philippe Cuvillier and Arshia Cont. Coherent time modeling of semi-markov models with application to real-time audio-to-score alignment. In *MLSP*, 2014.

[38] Cycling'74. The max-test package.

[39] R. Dannenberg. Music representation: A position paper. In *Proceedings of the International Computer Music Conference*, pages 73–75, San Francisco, USA, 1989. ICMA.

[40] Roger B. Dannenberg. Machine tongues xix: Nyquist, a language for composition and sound synthesis. *Computer Music Journal*, 21(3):71–82, 1997.

[41] P. R. D'Argenio, J. P. Katoen, T. C. Ruys, and J. Tretmans. *The bounded retransmission protocol must be on time!*, pages 416–431. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[42] Pedro R. D'argenio, Joost pieter Katoen, Theo Ruys, and Jan Tretmans. Modeling and verifying a bounded retransmission protocol. In *COST 247, International Workshop on Applied Formal Methods in System Design*, 1996.

[43] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[44] Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Marius Mikucionis, and Brian Nielsen. Testing real-time systems under uncertainty. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *9th International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 6957 of *Lecture Notes in Computer Science*, pages 352–371. Springer, 2010.

[45] Pierre Donat-Bouillud, Jean-Louis Giavitto, Arshia Cont, Nicolas Schmidt, and Yann Orlarey. Embedding native audio-processing in a score following system with quasi sample accuracy. In *ICMC 2016 - 42th International Computer Music Conference*, Utrecht, Netherlands, September 2016.

[46] Jose-Manuel Echeveste. *A programming language for Computer-Human Musical Interaction*. Theses, Université Pierre et Marie Curie - Paris VI, May 2015.

[47] Léa Fanchon and Florent Jacquemard. Formal Timing Analysis Of Mixed Music Scores. In *2013 ICMC - International Computer Music Conference*, Perth, Australia, August 2013.

[48] Wang Ge, Perry R. Cook, and Spencer Salazar. Chuck: A strongly timed computer music language. *Computer Music Journal*, 39(4):10–29, 2015.

[49] M. Gerhold and M. I. A. Stoelinga. Ioco theory for probabilistic automata. In *Proceedings of the 10th Workshop on Model Based Testing, MBT 2015, London, UK*, volume 180 of *Electronic proceedings in theoretical computer science*, pages 23–40, London, April 2015. Open Publishing Association.

[50] Arkadeb Ghosal, Thomas A. Henzinger, Christoph M. Kirsh, and Marko A. A. Sanvido. *Event-Driven Programming with Logical Execution Times*. Springer Berlin Heidelberg, 2004.

[51] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[52] Yann Guédon. Hidden hybrid markov/semi-markov chains. *Computational Statistics & Data Analysis*, 49(3):663 – 688, 2005.

[53] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: An industrial case study using uppaal. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 2–, Washington, DC, USA, 1997. IEEE Computer Society.

[54] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84– 99, January 2003.

[55] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

[56] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Formal methods and testing. chapter Testing Real-time Systems Using UPPAAL, pages 77–117. Springer-Verlag, Berlin, Heidelberg, 2008.

[57] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[58] Florent Jacquemard and Clement Poncelet. An automatic test framework for interactive music systems. *Journal of New Music Research*, 45(2):87–100, 2016.

[59] David Jaffe. Ensemble timing in computer music. *Computer Music Journal*, 9(4):38–48, 1985.

[60] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[61] Henrik Ejersbo Jensen, Kim G. Larsen, Henrik Ejersbo, Jensen Kim, G. Larsen, Arne Skou, and Arne Skou. Modelling and analysis of a collision avoidance protocol using spin and uppaal, 1996.

[62] Christoph M. Kirsh and Ana Sokolova. *The Logical Execution Time Paradigm*. Springer Berlin Heidelberg, 2012.

[63] Moez Krichen and Stavros Tripakis. Model checking software: 11th international spin workshop, barcelona, spain, april 1-3, 2004. proceedings. pages 109–126, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[64] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 34(3):238–304, June 2009.

[65] Marta Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theor. Comput. Sci.*, 282(1):101–150, June 2002.

[66] EW Large and MR Jones. The dynamics of attending: How people track time-varying events. *Psychological Review*, 106:119–159, 1999.

[67] M. Laurson. *PatchWork: a visual programming language and some musical applications*. Studia musica. Sibelius Academy, 1996.

[68] Edward A. Lee, Stephen Neuendorffer, and Gang Zhou. Dataflow. In Claudius Ptolemaeus, editor, *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[69] Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, Dec 1997.

[70] Atif M. Memon. An event-flow model of gui-based applications for testing. *Softw. Test., Verif. Reliab.*, 17(3):137–157, 2007.

[71] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[72] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267 – 310, 1983.

[73] Nils Peters, Trond Lossius, and Timothy Place. An automated testing suite for computer music environments. In *9th Sound and Music Computing Conference (SMC)*, Copenhagen, DK, 11/07/2012 2012. Nominated for Best Paper Award.

[74] Clément Poncelet and Florent Jacquemard. Model Based Testing of an Interactive Music System. In *ACM SAC*, Salamanca, Spain, April 2015.

[75] Clement Poncelet and Florent Jacquemard. Model-based testing for building reliable realtime interactive music systems. *Science of Computer Programming*, 132, Part 2:143 – 172, 2016. Special Issue on Software Verification and Testing (SAC-SVT'15).

[76] Clément Poncelet Sanchez and Florent Jacquemard. Test Methods for Score-Based Interactive Music Systems. In *ICMC SMC 2014*, Athen, Greece, September 2014.

[77] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[78] Miller Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77, Winter 1991.

[79] Miller Puckette. Pure data: Recent progress. In *Proceedings of the Third Intercollege Computer Music Festival*, pages 1–4, 1997.

[80] Antoine Rollet. Model based testing : principes et applications dans le cadre temporisé. *Université de Bordeaux (LaBRI - CNRS UMR 5800)*, 2011.

[81] Robert Rowe. *Interactive Music Systems: Machine Listening and Composing*. MIT Press, Cambridge, MA, USA, 1992.

[82] Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '08, pages 250–264, Berlin, Heidelberg, 2008. Springer-Verlag.

[83] Klaus Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms (Texts in Theoretical Computer Science. An EATCS Series)*. Springer, December 2003.

[84] Heinrich Taube. Common music: A music composition language in common lisp and clos. *Computer Music Journal*, 15(2), July 1991.

[85] Mark Timmer, Ed Brinksma, and Mariëlle Stoelinga. Model-based testing. In *Software and Systems Safety - Specification and Verification*, volume 30 of *NATO Science for Peace and Security Series - D: Information and communication Security*, pages 1–32. IOS Press, 2011.

[86] D.S. Touretzky. *Common LISP: A Gentle Introduction to Symbolic Computation*. Dover Books on Engineering. Dover Publications, 2014.

[87] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[88] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.

[89] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Formal methods and testing. chapter Model-based Testing of Object-oriented Reactive Systems with Spec Explorer, pages 39–76. Springer-Verlag, Berlin, Heidelberg, 2008.

[90] Ge Wang and Perry R. Cook. On-the-fly programming: Using code as an expressive musical instrument. In *Proceedings of the 2004 Conference on New Interfaces for Musical Expression*, NIME '04, pages 138–143, Singapore, Singapore, 2004. National University of Singapore.

Appendix A

Byte-Code IRTM

We present an example of a byte-code representation from a IRTM presented in the Section 5.2.2. The code is depicted in Figure A.1. A line contains: the instruction index in the code and the transition name or a list of transitions in case of branches. When an arrow is depicted after an instruction, the succeeding number is the index of the next instruction, if not depicted it is the instruction just after. We keep the event symbols in the listing to understand the code, however, it can be mapped in simple signal numbers. Finally, the $\|$ denotes an *and*-transition, the two numbers corresponds to the next locations ℓ_1 and ℓ_2.

```
000  and  -> 001 || 013
001  and  -> 002 || 009
002  emit e0
003  wait [1]
004  emit e1
005  wait [1]
006  emit e2
007  wait [2]
008  branch
 receive 2 -> 008

009  receive e0
010  receive e1
011  receive e2
012  stop
013  and  -> 014 || 026
014  and  -> 015 || 036
015  stop
016  emit a0
017  wait [0.5]
018  emit a1
019  wait [0.5]
020  emit a2
021  wait [0.5]
022  emit a3
023  stop
024  branch
 receive mSe1 -> 025

 receive e1 -> 020

025  stop
026  branch
 receive e0 -> 027

 receive mSe0 -> 028

027  and  -> 029 || 016
028  and  -> 030 || 024
029  stop
030  stop
031  emit a4
032  wait [0.5]
033  emit a5
034  stop
035  stop
036  branch
 receive e1 -> 037

 receive mSe1 -> 038

037  and  -> 039 || 031
038  and  -> 040 || 035
039  stop
040  stop
```

Figure A.1: Byte-code listing for the example Section 5.2.2.

List of Figures

List of Tables