

An Automatic Test Framework for Interactive Music Systems

Florent Jacquemard, Clément Poncelet

► **To cite this version:**

Florent Jacquemard, Clément Poncelet. An Automatic Test Framework for Interactive Music Systems. *Journal of New Music Research*, Taylor & Francis (Routledge), 2016, 45 (2), pp.18. <<http://www.tandfonline.com/doi/full/10.1080/09298215.2016.1173707>>. <hal-01274035>

HAL Id: hal-01274035

<https://hal.inria.fr/hal-01274035>

Submitted on 15 Feb 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Automatic Test Framework for Interactive Music Systems *

Florent Jacquemard and Clement Poncelet

Sorbonne Universités, Inria, UPMC Univ Paris 06, IRCAM – CNRS UMR SMTS, Paris, France

Abstract

Score-Based Interactive Music Systems are involved in live performances with human musicians, reacting in realtime to audio signals and asynchronous incoming events according to a pre-specified timed scenario called mixed score. Building such a system is a difficult challenge implying strong requirements of reliability and robustness to unforeseen errors in input.

We present the application to an automatic accompaniment system of formal methods for conformance testing of critical embedded systems. Our approach is fully automatic and based on formal models constructed directly from mixed scores, specifying the behavior expected from the system when playing with musicians. It has been applied to real mixed scores and the results obtained have permitted to identify bugs in the tested system.

Keywords: model based conformance testing, realtime systems, score based interactive music systems, generation of artificial music performances.

Introduction

Interactive music systems (IMS) [23] play in live music performances with human musicians. They work by coupling functionalities of artificial listening, for score following and tempo detection, and of reactive systems, for coordinating their outputs with musician inputs. In the case of score-based IMS, all these activities are performed in realtime, following a pre-specified timed scenario called a *mixed score*, written in a Domain Specific Language (DSL). It defines the parts of human musicians (input) together with electronic parts (output) and their synchronization.

During an instrumental performance, when a musician does a mistake, the piece must and will continue. However, IMS practitioners know that a misbehavior of an IMS can jeopardize a mixed instrumental-electronic performance. In order to build reliable IMS and meet listeners' expectations, it is important to be able to explore, statically, the IMS reactions to as many as possible musician's interpretations, and check that these reactions conform to the behavior specified in the given mixed score. This difficult task is complicated by high unpredictability of musicians' inputs and hard temporal constraints (due in particular to the strong requirements of audio computing platforms).

A traditional manual approach consists in

*This work has been partly supported by a DGA-MRIS scholarship and the project Inedit (ANR-12-CORD-009).

rehearsing with musicians. However time is precious during a rehearsal, whose purpose is usually more to solve musical problems than to fix bugs. It is also possible for developers to listen to recordings of an IMS playing with some musicians, checking a posteriori that the result *sounds* satisfiable. The problem with this approach is that, on the one hand, the test input is not complete (it just represents one or a few particular performances) and has to be played entirely, so such a testing procedure is tedious and time consuming. On the other hand the verification of the outcome is not rigorous.

This paper presents a study of the application of Model-Based Testing (MBT) methods to the score based IMS Antescofo, developed at Ircam and used regularly in concerts. This system shares several characteristics with the critical systems traditionally targeted by MBT, such as reactivity and real-time semantics.

The main originality of our contribution is the automatic construction of a formal model \mathcal{M} of the system's IO behavior from a given mixed score, using a front-end compiler into an ad hoc *intermediate representation* (IR). In this approach, the score is seen as a specification of the expected system's behavior. This is in contrast with usual MBT case studies where the model has to be built manually by an expert. On the base of the constructed model, we address the following issues:

- (i) exhaustive generation of test data for input, including timing values (*artificial performances*), following covering criteria, and applying model-checking techniques to \mathcal{M} ,
- (ii) computation of the corresponding expected output data, according to the input and the model, hence the mixed score,
- (iii) black-box execution of the generated test data using *virtual clocks* in order to play in a fast-forward mode. The outcome of the system is then compared formally to the expected outputs to check a Relativized Timed Input/Output Conformance relation (rtioco) between traces and a user readable verdict is produced.

Although the paper present an application to a specific IMS, our framework is based on a generic model (the IR), and could therefore be applied to other *score-based* IMS using, like Antescofo, a reactive DSL (relating discrete output to discrete input, with time values). Providing a compiler from a DSL into our IR (such as our adhoc front-end compiler from Antescofo DSL into IR), one could reuse our framework for the above steps (i) and (ii) without modifications. Note also that the steps of our workflow are independent, and linked using a generic timed trace format.

Structure of the paper. In the first part of the paper (Section 1), we present our model-based testing approach from the point of view of the user, considered as an external observer. We first present briefly our target system Antescofo and its DSL for writing mixed scores (1.1, 1.2). Then we define the format of the test data (*traces*), in input and output (1.4), and describe our testing workflow (1.3, 1.5, 1.6).

The second part (Section 2) details the internals of our test procedure, *i.e.* the models of the system's behavior (2.1) and how they are used to generate test data. We use the symbolic model checker Uppaal [14] and its extension CoVer [6], both based on the standard model of *Timed Automata* [3], for the production of test input and output data, with some covering criteria (2.2). We also propose alternative techniques for producing test input data: fuzzing of an ideal trace using performance models of the literature (2.3), and the generation of input data from audio recordings (2.4).

Finally, we present (Section 3) some relevant experimental case studies, the first one based on a benchmark of several hundreds of small mixed scores, and the other based on an extract of the real piece of mixed music *Einspielung* by Emmanuel Nunes.

Related Work. Some tools exists for automating the test of IMS, like for instance the max-test package [18] for testing MAX

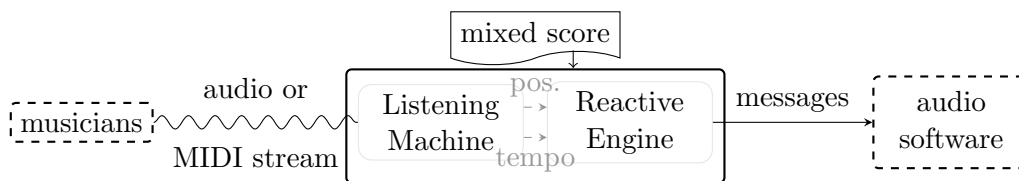


Figure 1: Architecture of Antescofo

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 ([19, 20]) 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 work has addressed the formal verification of multimedia systems based on Timed Automata models, such as for instance the verification of a lip-synchronisation protocol (synchronization of audio and video streams) in [7]. Timed Automata Networks, Uppaal, as well as timed Petri nets, are used in i-Score [5], 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 Interactive Music Systems.

1 IMS Test Framework

We introduce in this section our case study, the score follower Antescofo (1.1), its domain specific language (DSL) for writing mixed scores (1.2), our Model-Based Testing (MBT) workflow (1.3), whose main steps are the generation of test cases, the execution on the system under test (1.5) and test verdict (1.6). Test cases are timed traces, which is the format for data exchange in our test framework (1.4). Their generation from models is presented in Section 2.

1.1 The IMS Antescofo

Figure 1 describes roughly the architecture of Antescofo, which is made of two main modules. A *listening machine* (LM) decodes an audio or midi stream incoming from a musician and infers in realtime: (i) the musician’s position in the given mixed score, (ii) the musician’s instantaneous pace (*tempo*, in beats per minute) [8]. These values are sent to the second module: a *reactive engine* (RE) which schedules the electronic actions to be played, as specified in the mixed score. The actions are messages (*i.e.* instructions) emitted *on time* to an audio environment: a realtime audio software such as MAX/MSP [21] or Pure Data [22], in which Antescofo is embedded as a patch.

Therefore, from a behavioral point of view, the RE can be seen as a reactive system receiving and sending discrete events: some *input events* sent by the LM to the RE and *output events* sent by the RE to the environment. We propose a uniform format for these events in Section 1.4.

1.2 Domain Specific Language

The mixed scores of Antescofo are written in a textual language allowing the description of the electronic accompaniment in reaction to the detected instrumental events. A simplified extract of the score of *Einspielung I*¹ by Emmanuel Nunes is presented in Figure 2. This piece for violin and electronics will be used as a running example in this paper.

¹<http://brahms.ircam.fr/works/work/32409/>

Example 1. Figure 2 displays the first bar of *Einspielung*'s mixed score in Antescofo DSL. The violin part is represented in common western music notation in Figure 3. The keywords **note** and **chord** in Figure 2 are used to represent the input events expected from the violin (**chords** represent double stops). They are followed by a note pitch (or list of pitches), a duration (always $\frac{1}{7}$ in the example) and a label (e_1, \dots, e_7). The electronic part is specified with actions and sequences of actions called *groups*. Each action is expressed with a delay (time to wait before throwing action) followed by a message (abstract symbols a_0, \dots, a_7 in the example). A group is expressed with a delay, a label (s_2 in the example) followed by a sequence of actions or nested groups.

The interleaving of notes/chords and sequences of actions and groups specifies the coordination between input events and output actions: a sequence of actions ($a_0, s_2 \dots$) is triggered by the previous note (e_1 in the example). A zero delay between a triggering note and the first triggered action means simultaneity: for instance both a_0 and the group s_2 are started (in this order) as soon as e_1 is detected. Note that the total duration of the group s_2 is bigger than the duration of the triggering event e_1 (1 vs $\frac{1}{7}$). It means

bpm	144				
note	D4	1/7	e_1		
	0	a_0			
group	0	s_2	@loose	@global	
	{				
		0	a_1		
		1/7	a_2		
		1/7	a_3		
		1/7	a_4		
		1/7	a_5		
		1/7	a_6		
		1/7	a_7		
	}				
chord	(B3 \flat D4)	1/7	e_2		
chord	(E4 D4)	1/7	e_3		
chord	(D5 \sharp D4)	1/7	e_4		
chord	(A4 D4)	1/7	e_5		
chord	(C4 \flat D4)	1/7	e_6		
chord	(G4 D4)	1/7	e_7		

Figure 2: The first bar of *Einspielung*'s mixed score in Antescofo DSL (simplified).

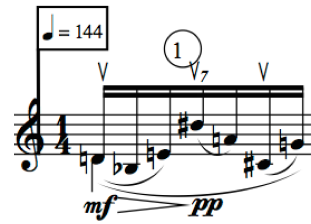


Figure 3: The first bar of *Einspielung*'s violin part in common western notation.

```

BPM 144      s1 = act(0, [a0], [ ]);
evt(e1, 1/7, s1);      act(0, s2, [loose; global])
evt(e2, 1/7, [ ]);    s2 = act(0, [a1], [ ]);
evt(e3, 1/7, [ ]);    act(1/7, [a2], [ ]);
evt(e4, 1/7, [ ]);    act(1/7, [a3], [ ]);
evt(e5, 1/7, [ ]);    act(1/7, [a4], [ ]);
evt(e6, 1/7, [ ]);    act(1/7, [a5], [ ]);
evt(e7, 1/7, [ ]) where act(1/7, [a6], [ ]);
                                     act(1/7, [a7], [ ])

```

Figure 4: Abstract syntax for the mixed score of Figure 2.

that the execution of this group will continue in concurrence with the detection of the next events e_2, \dots, e_7 . \diamond

The DSL of Antescofo offers an important set of features described in [12]. Composers can use different strategies of synchronization with the musicians' inputs for accompaniment, and different errors management strategies *etc.* We present here a simplified *abstract syntax* corresponding to a fragment of this language, in order to illustrate our test framework.

Let \mathcal{O} be a set of *output messages* (also called *action symbols* and denoted a) which can be emitted by the system and let \mathcal{I} be a set, disjoint from \mathcal{O} , of *event symbols* (denoted e) to be detected by the LM (*i.e.* positions in score). An *action* is a term $\text{act}(d, s, al)$ where d is the delay before starting the action, s is either an atom in \mathcal{O} or a finite sequence of actions, and al is a list of attributes. A *mixed score* is a finite sequence of *input events* of the form $\text{evt}(e, d, s)$ where

$e \in \mathcal{I}$, d is a duration and s is the *top-level group* triggered by e . Sequences are denoted with square brackets $[\]$ and the empty sequence is denoted $[]$. Figure 4 depicts the running example in the abstract syntax.

The high-level attributes attached to an action $\text{act}(d, s, al)$ are indications regarding musical expressiveness [9, 12]. We consider here four attributes for illustration purpose, two attributes are used to express the synchronization of the group s to the musician's part: **loose** (synchronization on tempo) and **tight** (synchronization on events) and two attributes describe strategies for handling errors in input: **local** (skip actions) and **global** (play actions immediately at the detection of an error).

We define an *error* as an event of the score missing during the performance, either because the musician played a wrong note, did not play it at all or because it was not detected by the LM.

Example 2. Figure 11 illustrates Antescofo's behavior for various combinations of attributes for the group s_2 of our running example. Note that in Figures 2 and 4, the attributes **loose** and **global** are selected. \diamond

Allowing the user to express error handling strategies and different degrees of smoothness for the system reactions are interesting features in a context of music composition but can complicate the understanding of the

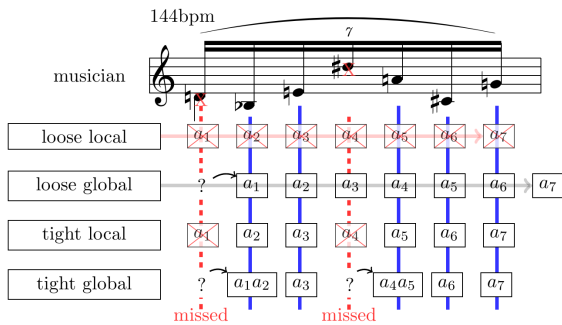


Figure 5: Reactions for various combinations of attributes for s_2 , when the events e_1 and e_4 are missed.

mixed scores and their validation.

1.3 Test Workflow

A mixed-score is seen in our case as a specification describing precisely the timed behavior expected from the system during a performance. It expresses the durations of input events (from the musician) corresponding to an *ideal* performance. In practice, real performances are not (and should not be) ideal: the tempo of the musician will be fluctuating, some notes' durations can be shifted and accidents can also occur (missing notes). The set of real performances is then infinite and no two performances will be the same. The purpose of a good test procedure is to assess the response of the system to a representative set of performances, as covering as possible.

Several formal methods have been developed for automatic conformance testing of critical embedded software, see *e.g.* [17]. Here we follow a Model-Based Testing (MBT) approach, where a formal model is used to construct representative performances and expected system's answers. Tests are executed with a real *implementation under test* (IUT), seen as a *black-box* (the source code of the IUT is not known and only the inputs and outputs are observed).

More precisely, in conformance MBT methods, a formal model \mathcal{M} of the system is needed. In general it is written manually by an expert. In our case however, it is obtained automatically by *compilation* of the mixed score, which is assumed to contain sufficiently information about the behavior expected from the system. Indeed, by essence,

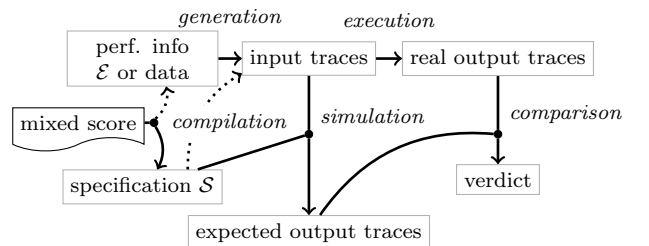


Figure 6: MBT workflow for IMS.

a mixed score describes precisely the relation between the outputs of the system and musician's events, with timing information. Such a scenario is very convenient in a context of assistance to composers of musical productions, who are usually subject to strict calendar constraints.

The model (see Section 2.1) is composed of two parts: \mathcal{S} the specification, describing the behavior expected for the IUT, and \mathcal{E} the environment, defining the inputs possible during the tests. The *Model* is used (Figure 6 and Section 2.2) to generate automatically some relevant test data: the input trace t_{in} , sent to the IUT, which must conform to \mathcal{E} , and the theoretically expected output trace t_{out} , computed from t_{in} by *simulation* using \mathcal{S} . On the other hand, an *execution* of the IUT on t_{in} permits to monitor real output trace t'_{out} , which is then *compared* to t_{out} in order to produce a test verdict.

1.4 Input and Output Test Data

For realtime systems such as communication protocols, transportation control *etc.*, as for IMS, time is a semantic issue, not just a measure of efficiency. Therefore, the test traces t_{in} and t_{out} must be timestamped.

We consider two time units here: *physical time*, in seconds, and *musical time*, expressed in number of beats relatively to a tempo. Let us assume a *tempo curve* τ associating an instant tempo value, in beats per minute (bpm), to each date t (in physical time). The conversion of a duration d from musical time into physical time is obtained by integration in $[0, d]$ of the inverse of τ . Here, we shall always consider tempo curves which stay constant between the occurrences of two events. This corresponds to the assumption (used in Antescofo LM) that we infer tempo variations only at the arrival of events, and not in-between.

A *timed trace* is a sequence of triples $\langle a_i, t_i, p_i \rangle$ made of a symbol $a_i \in \mathcal{I} \cup \mathcal{O}$, a timestamp $t_i \in \mathbb{R}^+$ (*onset*), expressed in musical time, and a tempo value p_i in bpm,

such that for all i , $t_i \leq t_{i+1}$ and if $t_i = t_{i+1}$ then $p_{i+1} = p_i$. The tempo curve τ associated to such a trace is defined by $\tau(t) = p_i$ iff $t_i \leq t < t_{i+1}$. A trace containing symbols exclusively in \mathcal{I} (resp. \mathcal{O}) is called an *input trace* (resp. an *output trace*). We denote below \mathcal{T}_{in} (resp. \mathcal{T}_{out}) the set of input (respectively output) traces with timestamps in musical time (t_{beat}). The *ideal trace* for a mixed score S is the input trace of \mathcal{T}_{in} consisting in the projection of all input events in S with their date (accumulated duration) and the tempo given in S .

Example 3. The ideal trace for the score in Figure 2 is:

$$\langle e_1, 0, 144 \rangle \cdot \langle e_2, \frac{1}{7}, 144 \rangle \cdot \dots \cdot \langle e_6, \frac{5}{7}, 144 \rangle \cdot \langle e_7, \frac{6}{7}, 144 \rangle. \quad \diamond$$

Let us consider the two parts of model defined above \mathcal{E} as a set of possible input traces and \mathcal{S} as a simulation function from input traces to output traces. A *test case* is a pair $\langle t_{\text{in}}, t_{\text{out}} \rangle \in \mathcal{T}_{\text{in}} \times \mathcal{T}_{\text{out}}$ where $t_{\text{in}} \in \mathcal{E}$ is an input trace (an artificial performance) and $t_{\text{out}} = \mathcal{S}(t_{\text{in}})$ is the corresponding expected system's reaction. The automatic and covering generation of input traces t_{in} and expected output traces t_{out} from scores is the topic of Section 2.

Example 4. Let us consider the three following input traces for the piece in Figure 2.

$$\begin{aligned} t_{\text{in}}^1 &= \langle e_1, 0, 114 \rangle \cdot \langle e_2, \frac{1}{7}, 117 \rangle \cdot \langle e_3, \frac{2}{7}, 120 \rangle \cdot \\ &\quad \langle e_4, \frac{3}{7}, 117 \rangle \cdot \langle e_5, \frac{4}{7}, 114 \rangle \cdot \langle e_6, \frac{5}{7}, 111 \rangle \cdot \\ &\quad \langle e_7, \frac{6}{7}, 114 \rangle. \\ t_{\text{in}}^2 &= \langle e_1, 0, 114 \rangle \cdot \langle e_3, \frac{3}{14}, 120 \rangle \cdot \langle e_4, \frac{6}{14}, 117 \rangle \cdot \\ &\quad \langle e_6, \frac{9}{14}, 111 \rangle \cdot \langle e_7, \frac{12}{14}, 114 \rangle. \\ t_{\text{in}}^3 &= \langle e_1, 0, 114 \rangle \cdot \langle e_2, \frac{9}{70}, 117 \rangle \cdot \langle e_3, \frac{18}{70}, 120 \rangle \cdot \\ &\quad \langle e_4, \frac{27}{70}, 117 \rangle \cdot \langle e_5, \frac{36}{70}, 114 \rangle \cdot \\ &\quad \langle e_6, \frac{45}{70}, 111 \rangle \cdot \langle e_7, \frac{54}{70}, 114 \rangle. \end{aligned}$$

In trace t_{in}^1 the durations are those of the mixed score but the tempo diverges from the ideal tempo (unlike Example 3). In t_{in}^2 , the onsets are shifted (to play the events a bit earlier or later) and events e_2 , e_5 are missed. In the last trace t_{in}^3 all events are played 10% earlier than in t_{in}^1 . \diamond

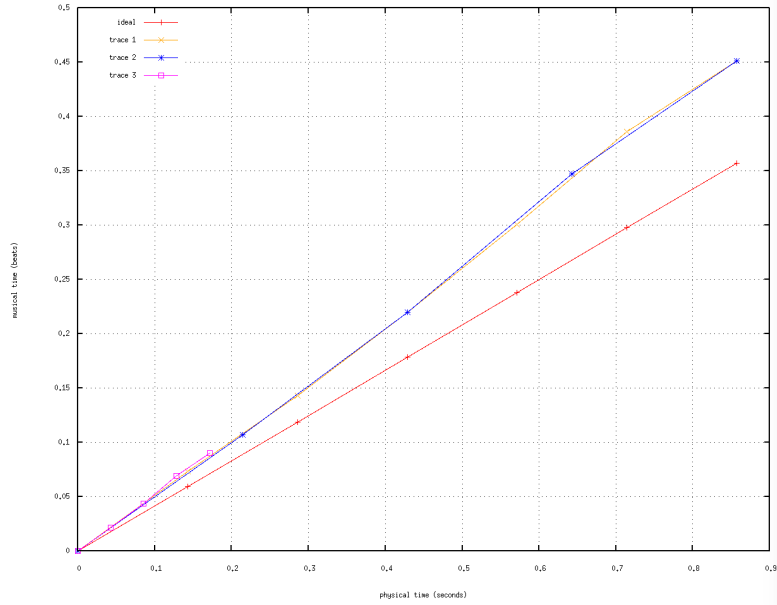


Figure 7: Input traces t_{in}^1 , t_{in}^2 and t_{in}^3 , physical values according relative ones.

1.5 Test Execution

Having defined a format for timed traces, the next question is: *How can we observe the reaction of the system to a given input?*

The execution of an input trace t_{in} is somehow a monitored simulation of a performance. It consists in sending the events in t_{in} to the IUT, at the specified dates, and collect a *real output trace* t'_{out} by monitoring and time-stamping all output emitted by the system during the execution.

The problem can be complex due to the data flow in Antescofo, its modular nature (Figure 1) and the variety of time units that can be used in timed traces. We present below several scenarios for testing different parts of the system corresponding to different boundaries for the black box tested.

1.5.1 Testing the Reactive Engine

This scenario 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 t_{in} in a file. The duration $d_i^{mu} = t_{i+1} - t_i$ of the event e_i (in musical time) is converted into physical

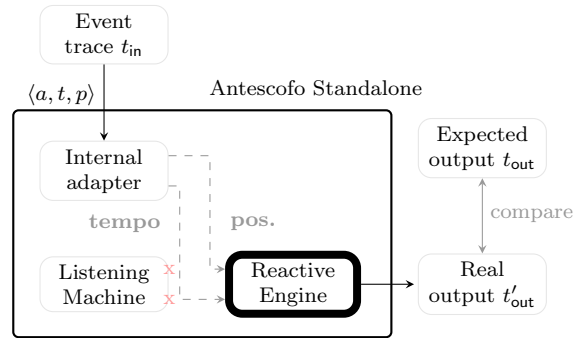


Figure 8: Testing scenario of Section 1.5.1.

time by:

$$d_i^{ph} = \frac{d_i^{mu} \cdot 60}{p_i} \quad (1)$$

The adapter then waits for d_i^{ph} 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^{ph} 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.

The messages sent by the RE are logged in t'_{out} , with timestamps in physical time (*i.e.* with a tempo of 60bpm). In this scenario, the

IUT is the RE (the LM is idle).

1.5.2 RE with Tempo Detection

In this second scenario, tempo values p_i read in t_{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^{ph} . The rest of the scenario is similar to Section 1.5.1. 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 t_{out} from musical to physical time, in order to be able to compare it with the real output trace t'_{out} . In this case, the IUT consists in the RE plus the part of the LM in charge of tempo inference. The LM infers the tempo based on the shift between durations in t_{in} and in the mixed score [8]. It might result in a tempo increasing exponentially when durations in t_{in} are too short.

Example 5. Let us see how the detected tempo can increase by executing our trace t_{in}^3 via this scenario. The duration of e_1 is computed with the timestamp of e_2 found in t_{in}^3 : $d_1^{mu} = \frac{9}{70} - 0$. Then we obtain a physical value of $d_1^{phy} = 0.05357$ seconds with a tempo of 144 (score value by default). The detection of e_2 is earlier than expected (since the relative time is shorter than 10%), Antescofo's LM modifies its current tempo to 146bpm. The computation of the same rela-

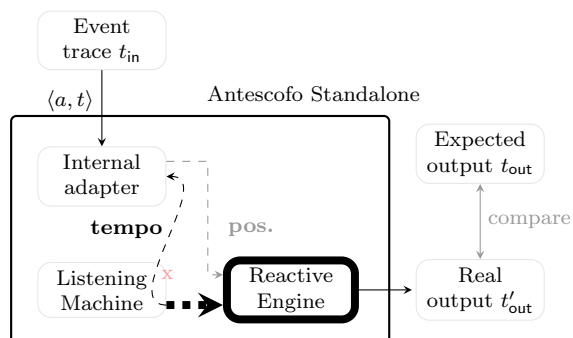


Figure 9: Testing scenario of Section 1.5.2.

tive duration for the next event is done with a faster tempo and gives $d_2^{phy} = 0.05283$ sec, the event is earlier so the next tempo is faster than 146bpm and so on. In this very short example (and with small shifted durations), we reach at the end a tempo of 150.3bpm, that is 6.3bpm more than the score tempo in 0.4 seconds of a performance. \diamond

1.5.3 Testing the Whole System

This scenario is the most general. It is executed with a version of Antescofo embedded in MAX (as a MAX patch), using an adapter which is another MAX patch. The adapter iteratively reads triples $\langle a_i, t_i, p_i \rangle$ in a file containing t_{in} , and converts them into MIDI events, with durations d_i^{ph} casted into physical time using (1). The events are played by the MAX patch `midisynth~` and the audio stream generated is sent to the LM. The output of the RE is then traced in t'_{out} as before.

Note that here, the RE uses the tempo values detected by the LM, which may differ from the tempo values in t_{in} . The detected tempo values are saved by the adapter (in MAX the detected tempo is available as an outlet of the `antescofo~` patch). They are used later to convert the dates in t_{out} from musical into physical time, like in Section 1.5.2. 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 t_{in} for computing the events' durations d_i^{ph} , like in Section 1.5.1.

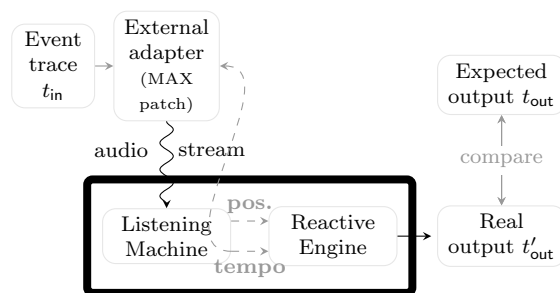


Figure 10: Testing scenarios of Section 1.5.3.

Note that in both scenarios of this section, the tests are executed in real-time and not in a fast-forward mode like in Sections 1.5.1 and 1.5.2. However an audio file of the sequence of MIDI sounds can be recorded and sent later to the standalone in fast-forward mode.

1.6 Comparison and Verdict

How can we check that the real output trace t'_{out} is correct? When the expected output trace t_{out} is not precisely defined, we are left to listen to the execution of the IMS on t_{in} , in extenso, for instance using the framework presented in Section 1.5.3, and decide subjectively whether we are satisfied with it. This manual solution is not precise and also tedious when one needs to consider many different t_{in} for covering purposes.

In our approach, we compute t_{out} from t_{in} , as described in Section 2, and the verdicts are produced offline by a tool comparing the expected output trace t_{out} to the monitored output trace t'_{out} (after conversion of timestamps into physical time as described above).

One difficulty for the comparison of traces is that some messages might be missing in t'_{out} or the order of close messages might be inverted. We do not use a simple componentwise comparison between t_{out} and t'_{out} , but instead compares the respective dates of each symbol in these two output traces. For the comparison we use a fixed tolerance bound δ , for dealing with latency. In the reported experiments, we have used $\delta = 0.1\text{ms}$.

Example 6. A running example is done to present an error case seen during an Antescofo's test. Let us consider the input trace t_{in}^4 derived from t_{in}^3 with the first event missing and the tempo divided by two. The beginning of the corresponding expected and real output traces (respectively t_{out}^4 and t'_{out}^4) are reported below and the verdict is depicted in Figure 12.

$$t_{\text{in}}^4 = \langle e_2, 0, 62 \rangle \cdot \langle e_3, \frac{9}{70}, 64 \rangle \cdot \langle e_4, \frac{18}{70}, 60 \rangle \cdot \langle e_5, \frac{27}{70}, 58 \rangle \cdot \langle e_6, \frac{36}{70}, 56 \rangle \cdot \langle e_7, \frac{45}{70}, 58 \rangle.$$

$$t_{\text{out}}^4 = \langle e_2, 0, 62 \rangle \cdot \langle a_0, 0, 62 \rangle \cdot \langle a_1, 0, 62 \rangle \cdot \langle a_2, 0, 62 \rangle \cdot \langle e_3, 0.12856, 64 \rangle \cdot \langle a_3, 0.14285, 64 \rangle \dots$$

$$t'_{\text{out}}^4 = \langle a_0, 0, 60 \rangle \cdot \langle a_1, 0, 60 \rangle \cdot \langle e_2, 0, 60 \rangle \cdot \langle e_3, 0.12442, 60 \rangle \cdot \langle a_2, 0.13781, 60 \rangle \cdot \langle e_4, 0.24495, 60 \rangle \cdot \langle a_3, 0.27353, 60 \rangle \dots$$

In the expected trace t_{out}^4 , the tempo values are copied from the input trace t_{in}^4 . Since the event e_1 is missed in t_{in}^4 , and according to the group attributes in Figure 3, a_0 , a_1 and a_2 are sent immediately when e_1 is detected as missing *i.e.* at the detection of e_2 . The delays in the real trace t'_{out}^4 are in physical time (60 bpm). Note that in this trace, a_0 and a_1 are before e_2 , contradicting the order in t_{out}^4 . However this does not alter the conformance because all these events are synchronous (time-stamp 0s). In contrast, the action a_2 is at 0 in t_{out}^4 and not in t'_{out}^4 , this is reported as an error in the verdict (Figure 12): the real trace t'_{out}^4 is not conform to t_{out}^4 , indicating a bug in Antescofo. \diamond

2 Models and Automatic Test Cases Generation

It remains to see *how to generate the test cases $\langle t_{\text{in}}, t_{\text{out}} \rangle$* . We present below several approaches based on models built from a given mixed score for creating a relevant and covering set of input traces t_{in} and computing the corresponding expected output traces t_{out} .

2.1 Models of Computation

The principle of MBT approaches is to rely on a formal model \mathcal{M} specifying the good behavior of the IUT. This model acts as an oracle computing the good outputs from a given input trace. As observed in Section 1.3, a mixed score describes formally the outputs of the system following musician's events (with timing information). Therefore, it is used in order to create automatically the model \mathcal{M} at the heart of our approach. More precisely, \mathcal{M} is constructed from an Abstract Syntax Tree (Figure 4), itself obtained by parsing a program in Antescofo DSL (Figure 2).

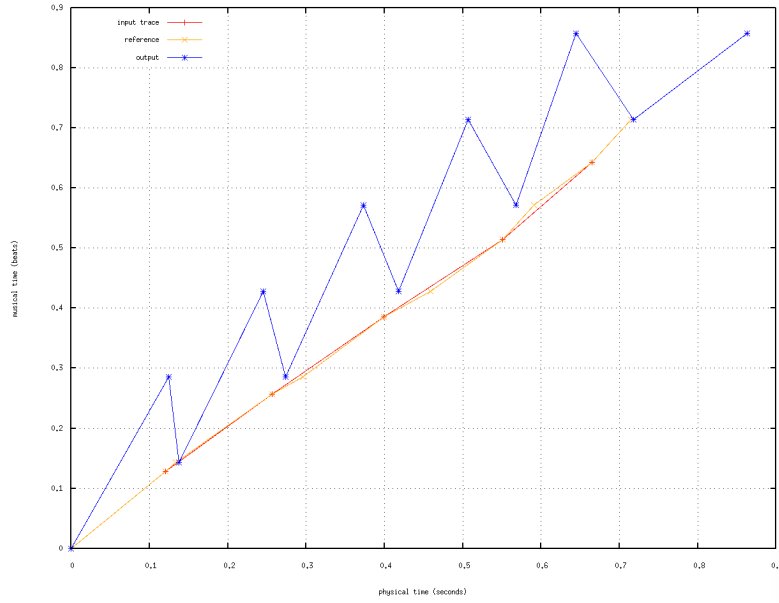


Figure 11: Traces t_{in}^4 , t_{out}^4 and t'^4 , physical values according relative ones.

2.1.1 Intermediate Representation

We use an Intermediate Representation (IR) for defining the formal models \mathcal{M} of our MBT framework. An IR program is a finite set (called *network*) of Finite State Machines (FSM) with durations, communicating synchronously with some symbols taken from a finite alphabet. Formally, an FSM is a tuple $S = \langle \Sigma_{in}, \Sigma_{out}, L, \ell_0, T \rangle$ where Σ_{in} and Σ_{out} are finite sets of respectively input and output symbols (they may have a non-empty intersection), L is a finite set of *locations*, $\ell_0 \in L$ is the initial location, $T \subseteq L \times L$ is a set of *transitions* labeled with one of:

$\sigma!$ with $\sigma \in \Sigma_{out}$ – emission of a symbol,

$\tau?$ with $\tau \in \Sigma_{in}$ – reception of a symbol,

$[d, d']$ where $d, d' \in \mathbb{R}_+$ are expressed in the same time unit (musical or physical) – wait for a delay between d and d' .

A *branch* is made of a location $\ell \in L$ and the set of transitions outgoing from ℓ . 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 $d = d'$ (in this case, we simply write d instead of $[d, d]$).

The execution of an IR program is based on a notion of logical time. Initially, the logical time is set to 0 and every FSM of the IR program is in its initial location ℓ_0 .

Activating in a FSM a transition tr labeled with $[d, d']$ makes the logical time advance by a duration between d and d' . More precisely, if the current logical instant is $t \in \mathbb{R}_+$, ℓ is the source location of tr and the logical time already spent in ℓ is γ , then the logical time can be advanced to the new current logical instant $t + \gamma + \delta$ if $\gamma + \delta \in [d, d']$.

Activating a transition labeled with $\sigma!$ does not change the current logical instant (such a transition is considered as instantaneous). The symbol σ is recorded in a store and will be readable during the current logical instant, but not after: the activation of a transition labeled with $[d, d']$ flushes the store.

A transition labeled with $\tau?$ can be activated if τ is present in the store. Such a transition does not change the current logical instant and neither changes the store.

Some examples of IR and their executions are given in Section 2.1.2.

Antescofo Trace				Expected Trace			
label	now	[rnow]		label	comp. timestamp	[ref beat]	
a0	0	[0.142857]		a0	0	[0]	
a1	0	[0.142857]		a1	0	[0]	
e2	0	[0.142857]		e2	0	[0]	* 62BPM
+ 0.124423 (0.142857 * 62)				> 0.124413 (0.12856 * 62)			
e3 0.124423		[0.285714]		e3 0.124413		[0.12856]	* 64BPM
+ 0.0133942 (0.0142871 * 64)				> -0.124413 (-0.12856 * 64)			
x a2 0.137817		[0.300001]		x a2 0		[0]	delta:0.137817
+ 0.10714 (0.12857 * 64)				> 0.107128 (0.11427 * 64)			
e4 0.244957		[0.428571]		e4 0.244938		[0.25712]	* 60BPM
+ 0.0285743 (0.0285743 * 60)				> -0.107128 (-0.11427 * 60)			
x a3 0.273531		[0.457146]		x a3 0.13781		[0.14285]	delta:0.135721
				[...]			
+ 0.0857229 (0.0857229 * 60)				> -0.0591 (-0.05713 * 60)			
-----END TIMESTAMP OF REF TRACE-----							
x a7 0.86301		[1.08572]		x a7 0.718148		[0.71425]	delta:0.144863

Error :: Test K0

Figure 12: Part of a verdict for the comparison of t_{out}^4 with t'_{out}^4 . The mark '+' indicates a new logical instant (see Section 2.1). The differences between the ideal trace and the input trace are shown with '<', '>' and '=='. The mark 'x' indicates an error.

2.1.2 Compiling Mixed Score into IR

For the sake of conciseness, we shall not describe in details the construction of models from mixed scores. We will instead explain its principles on some examples. The formal model \mathcal{M} is made of two parts: a FSM \mathcal{E} defining the possible behaviors of the environment (subset of \mathcal{T}_{in} , see Section 1.4), and a network \mathcal{S} specifying the behavior expected from the system (function from \mathcal{T}_{in} into \mathcal{T}_{out}). The sets of \mathcal{O} and \mathcal{I} are those of Section 1.2. We assume in addition a set \mathcal{Sigs} of symbols of *internal signals*, disjoint from \mathcal{O} and \mathcal{I} .

Environment. The environment \mathcal{E} is a non-deterministic FSM of the form $\langle \mathcal{O}, \mathcal{I}, L_e, \ell_0, T_e \rangle$.

Example 7. Figure 13 presents an example of environment model \mathcal{E} for the three first events of the piece in Figure 4. It models a musician which, in ℓ_0 , will possibly play the first note e_1 , or miss it (upper edge labeled with $e_2!$ targeted to ℓ_3) or miss e_1 and e_2 (edge labeled with $e_3!$ targeted to ℓ_5). When

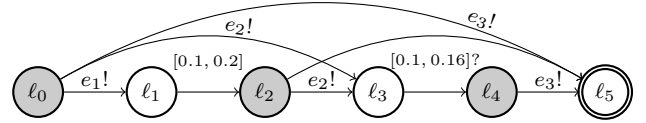


Figure 13: FSM for an environment \mathcal{E} modeling the 3 first events of the piece.

e_1 is not missed, this note must have a duration between 0.1 and 0.2 t.u. These bounds define a tolerance for negative or positive shifts to the duration of $\frac{1}{7}$ in the score. \diamond

For a given score, there are several options regarding missed notes and duration bounds (which will be source of non-determinism in \mathcal{E}). They are given by users in the command for compiling.

Error Proxy. In Section 1.2, we have defined the errors as missed events. An FSM of the form $\langle \mathcal{I}, \mathcal{Sigs}, L_p, \ell_0, T_p \rangle$, called a *proxy*, is in charge of signaling such errors, using one internal signal $\bar{e}_i \in \mathcal{Sigs}$ for each input event $e_i \in \mathcal{I}$. These signals are then handled by the other FSMs of \mathcal{S} .

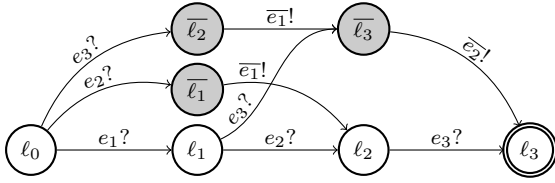


Figure 14: Proxy FSM \mathcal{P} with 3 events.

Example 8. In the proxy \mathcal{P} displayed in Figure 14, in location l_0 , at the detection of e_2 (instead of e_1), the signal \bar{e}_1 is sent. And at the detection of e_3 , the signals \bar{e}_1 and \bar{e}_2 are sent. \diamond

The use of this proxy FSM will simplify the complex task of specifying error handling in the other FSMs of the model \mathcal{S} . In particular, such a modular approach permits to change the definition of errors without having to change the rest of the specification \mathcal{S} .

FSM for Groups. Finally, we associate several FSM to the mixed score, one for each group. The FSM for a group s has the form $\mathcal{G}_s = \langle \mathcal{I} \cup \text{Sigs}, \mathcal{O} \cup \text{Sigs}, L_s, l_0, T_s \rangle$. It means that \mathcal{G}_s will receive input events and internal signals and send, in reaction, output messages and other internal signals.

Example 9. Figure 15 presents the FSM \mathcal{G}_{s_1} associated to the top-level group s_1 triggered by the event e_1 in the mixed score of our running example Figure 4. The FSM \mathcal{G}_{s_1} is started by the reception of one of the input symbol $e_1 \in \mathcal{I}$ or the internal signal $\bar{e}_1 \in \text{Sigs}$. In the first case, the FSM continues in a normal mode (location l_1). Otherwise it continues in an error mode (location \bar{l}_1). The FSM \mathcal{G}_{s_2} corresponding to the nested group s_2 (see Example 10) is started by the emission of one of the internal signals s_2 or \bar{s}_2 . \diamond

Example 10. Figures 16 and 17 show the FSM \mathcal{G}_{s_2} obtained from the group s_2 in Figure 4, for two different synchronization strategies (resp. loose and tight). Those models are constructed by iteratively traversing the sequence of actions in s_2 . The parts built

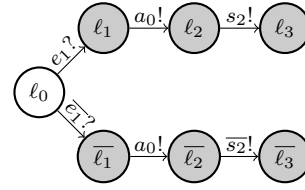


Figure 15: The FSM \mathcal{G}_{s_1} for the group s_1 .

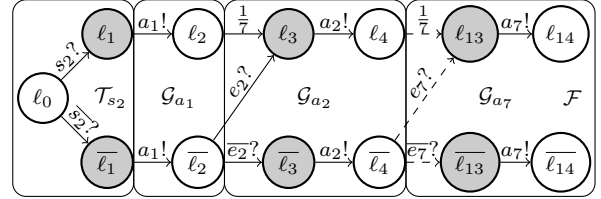


Figure 16: The FSM \mathcal{G}_{s_2} for the group s_2 with attributes loose, global.

at each step are framed and annotated as \mathcal{T}_g (start of the group, triggered by one of the internal signals s_2 and \bar{s}_2), \mathcal{G}_a (for handling one atomic action a), or \mathcal{F} (end of the group). The dashed transitions indicate a missing part of the representation (too long for being presented entirely).

In Figure 16 (attributes loose, global), the top part of the IR (locations l_1, \dots, l_{14}) describes a mode with a normal behavior (absence of errors). It consists in sending successively the actions after waiting for their respective delay (in musical time unit). The bottom part of the IR (locations $\bar{l}_1, \dots, \bar{l}_{14}$) describes the behavior in case of error: send instantaneously all actions (without delay) until the next detected event (in \bar{l}_2, \bar{l}_4).

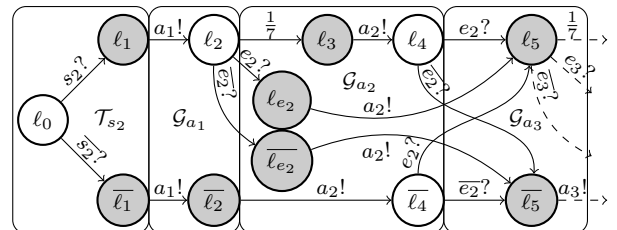


Figure 17: The FSM \mathcal{G}_{s_2} for the group s_2 with attributes tight, global.

In Figure 17 (attributes `tight`, `global`), the top and bottom parts also describe respectively normal and error modes. There is a possible switch (location ℓ_4) from the normal to the error mode if a missed event is detected. Also, in location ℓ_2 , if the event e_2 (or a signal \bar{e}_2 signifying its absence) is detected earlier than expected, then there is a switch to an intermediate mode, in location ℓ_{e_2} (respectively location $\bar{\ell}_{e_2}$). \diamond

The translation of the two strategies `loose` and `tight` of Antescofo DSL into FSMs are described here for illustration purpose. The approach is however more general: The target IR used for score compiling is quite generic, and should permit to compile other synchronization strategies (even from other DSLs).

Example 11. The simulation of the IR network $\mathcal{E} \parallel \mathcal{P} \parallel \mathcal{G}_{s_1} \parallel \mathcal{G}_{s_2}$, with \mathcal{G}_{s_2} from Figure 17, for the input trace $t_{\text{in}} = \langle e_1, 0.1, 60 \rangle, \langle e_3, -, 60 \rangle$ is presented as the following sequence of logical times separated by transitions:

$$0 \xrightarrow{e_1!} 0 \xrightarrow{e_1?} 0 \xrightarrow{a_0!} 0 \xrightarrow{s_2!} 0 \xrightarrow{s_2?} 0 \xrightarrow{a_1!} 0.1 \xrightarrow{e_3!} 0.1 \xrightarrow{e_3?} 0.1 \xrightarrow{\bar{e}_2!} 0.1 \xrightarrow{\bar{e}_2?} 0.1 \xrightarrow{a_2!} 0.1 \xrightarrow{a_3!} 0.1.$$

The expected output trace t_{out} is: $\langle a_0, 0, 60 \rangle \cdot \langle a_1, 0, 60 \rangle \cdot \langle a_2, 0.1, 60 \rangle \cdot \langle a_3, 0.1, 60 \rangle$. \diamond

2.2 Covering Test Case Generation

We use the model checker `Uppaal` in order to generate relevant test cases automatically from the IR models. The generation is guided by the part \mathcal{E} of the model which specifies the possible set of performances in the tests.

2.2.1 Timed Automata

`Uppaal`² is a symbolic model checker enabling to write, simulate and verify Timed Automaton Networks. Timed automata (TA) [3] are finite state automata extended with a finite set of real-valued variables called *clocks*. Every TA transition is labeled by a symbol (in a finite alphabet), and a linear constraint

(*guard*) on the clock values: the transition can be fired only if the current values of the clocks satisfy the associated constraint. Moreover, every clock can be independently reset to 0 during a transition and keeps track of the elapsed time since the last reset. Some linear constraints on the clock values called *invariants* can also be attached to states. Such a constraint must be satisfied as long as the control stays in the associated state. In a TA, all the clock values are expressed in a unique abstract time unit, the *model time unit* (mtu), *i.e.* all the clocks evolve at the same rate.

The set of configurations of a TA \mathcal{A} is infinite (it is the Cartesian product of the finite set of states of \mathcal{A} and the infinite set of valuations of the clocks of \mathcal{A}). However, it is possible to transform a TA into a finite state automaton recognizing the same (untimed) sequences of symbols, using a finite equivalence on configurations (*region construction*) [3]. This fundamental technique gives a PSPACE algorithm for deciding reachability properties, implemented efficiently in `Uppaal`.

From IR into TA. The above IR programs $\mathcal{E} \parallel \mathcal{S}$ can be translated into an equivalent TA network $\mathcal{A}_{\mathcal{E}} \parallel \mathcal{A}_{\mathcal{S}}$. However, because of differences in semantics, the translation is not totally immediate, and some adaptations were necessary to handle the synchronization of symbols during a logical instant.

The IR semantics was defined in order to conform to the semantics of Antescofo DSL [11, 12] and slightly differ from TA semantics. Indeed, thanks to the use of a store (Section 2.1.1), all the symbols emitted during a logical instant in an IR can be received afterwards during the same logical instant. This way, the reception of a symbol is not necessary synchronous with its emission. In contrast, in a TA network, two transitions labelled with the same symbol must be fired simultaneously. In order to simulate with TA the delayed reception of IR, some locations and transitions have to be added during the conversion.

²<http://www.uppaal.org>

2.2.2 Generation of Input Traces

We use the Uppaal extension called CoVer [14] to generate automatically suites of test cases, for certain \mathcal{E} and \mathcal{S} , according to some coverage criteria. These criteria are defined by a finite state automaton \mathcal{Obs} called *observer* monitoring the parallel execution of the TA network $\mathcal{A}_{\mathcal{E}}\|\mathcal{A}_{\mathcal{S}}$. Every transition of \mathcal{Obs} is labeled by a predicate checking whether a transition of $\mathcal{A}_{\mathcal{E}}\|\mathcal{A}_{\mathcal{S}}$ is fired. The model checker Uppaal is used by CoVer to generate the set of input traces $t_{\text{in}} \in \mathcal{T}_{\text{in}}$ resulting from an execution of the Cartesian product of $\mathcal{A}_{\mathcal{E}}\|\mathcal{A}_{\mathcal{S}}$ with \mathcal{Obs} reaching a final state of \mathcal{Obs} .

For instance, we can design an observer in charge of visiting all transitions corresponding to a missed event in the proxy, or visiting all the transitions of a particular group that we want to debug.

For loop-free IR \mathcal{S} and \mathcal{E} , with an observer checking that all transitions of $\mathcal{A}_{\mathcal{E}}\|\mathcal{A}_{\mathcal{S}}$ are fired, CoVer will return a test suite \mathcal{T} *refutationally complete* for conformance, in the sense that: if there exists an input trace $t_{\text{in}} \in \mathcal{E}$ such that $t'_{\text{out}} = \text{IUT}(t_{\text{in}})$ and $t_{\text{out}} = \mathcal{S}(t_{\text{in}})$ differ, then \mathcal{T} will contain such an input trace. Note that the IR produced by the fragment of the DSL of Section 1.2, are loop-free. However this is not true for the general DSL which allows *e.g.* jump to label.

In practice, we avoid state explosion with appropriate restrictions on \mathcal{E} , such as described in Section 2.1.2 (number of missed notes, bounds on event's durations). This way we generate a suite of tests cases covering the maximum of the \mathcal{Model} transitions according to environment model's bounds.

2.2.3 Computing Expected Output

The computation of the expected output trace $t_{\text{out}} = \mathcal{S}(t_{\text{in}})$ is done by simulation with Uppaal, using the translation of the IR into TA described Section 2.2.1.

More precisely, given t_{in} we first generate a deterministic FSM $\mathcal{E}_{t_{\text{in}}}$ which will strictly follow the input trace. This FSM is con-

verted into a deterministic TA $\mathcal{A}_{t_{\text{in}}}$. The simulation of the TA network is then performed by traversing the TA $\mathcal{A}_{t_{\text{in}}}$ and sending event symbols to the rest of the Model $\mathcal{A}_{\mathcal{S}}$. Uppaal offers options to trace the result in t_{out} .

2.3 Fuzzing Ideal Performances

We consider techniques for building artificial performances for test purposes (*i.e.* create test input traces t_{in}) by fuzzing ideal traces using models of performance defined in former works [1, 2, 10, 4, 15, 16].

2.3.1 Models of Performances

We have seen in Section 1.5 that we can convert an input trace t_{in} into a performance, *i.e.* a sequence of real durations, as follows. First, we extract from the third components of the triples in t_{in} a tempo curve τ , as described in Section 1.4. Second, we convert the durations, expressed in musical time in the second component of t_{in} 's triples, into real durations (in physical time) using τ . Note that when the two first components of triples in t_{in} corresponds to the events and durations specified in the score, the above performance is completely defined by the tempo curve τ .

Some works in musical cognitive research have proposed more accurate representation of musical performances. *Time-maps* by Jaffe [1], *time-warps* by Dannenberg [2], or *time-deformations* by Anderson and Kuivila [4] are monotonically non-decreasing functions mapping score durations (in musical time) into performed durations (in physical time). In [10], Dannenberg gives two special cases of such functions called *shift* and *stretch* operators. The first express operations such as delay, rest or pause and the second deals with the tempo variations.

In [15, 16], Honing proposes *Timing Functions* (TIF) which combine two time-warps: a tempo curve f^{\times} and a time-shift function f^{+} , defining variations of events' durations, independently of the tempo changes. Although tempo variations induce changes of durations

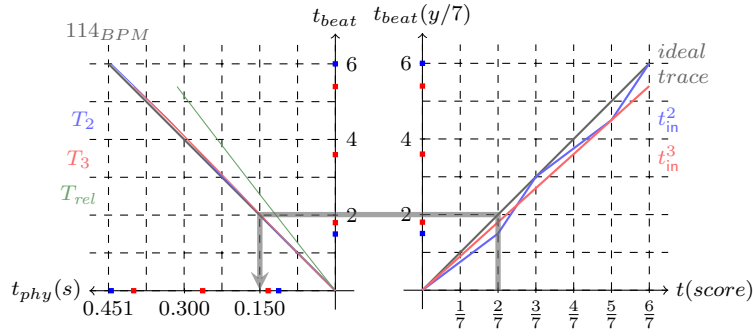


Figure 18: Input traces using TIFs. From the bottom right to the bottom left, the picture presents the application of two functions (the left one is rotated to 90° to the left to take in input the output of the first-right function). The three traces are so first shifted by the related color function and the shifted values are next translated with a tempo curve function.

and reciprocally, Honing outlines the interest of considering independently tempo curves and time-shifts for defining musical performances. They have indeed two well distinct musical significance. Roughly, the first describes global continuous changes of durations, and the second local changes (like *swing notes*).

2.3.2 Generation of Inputs Traces

In our MBT method, we construct input test traces using an extended TIF model that will transform (*fuzz*) the ideal trace associated to the given mixed score (as defined in Section 1.4) into an artificial performance t_{in} , see Figure 19. More precisely, the transformation applies the function f^+ which modifies the durations, then the tempo curve f^\times , and finally choose some missed events. This latter step is an addition of the models of [16, 10].

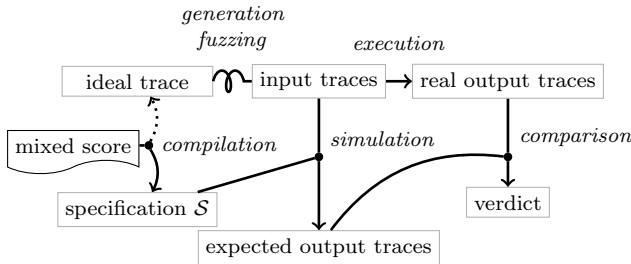


Figure 19: Fuzz testing workflow.

Example 12. We show in Figure 18 the given input traces of the previous section with TIFs. The gray data depicts ideal values, the blue t_{in}^2 and the red t_{in}^3 trace. \diamond

The implemented fuzzing function takes in input an ideal trace and three parameters for bounding the deviations on the time-shifts, the tempo values and the number of missing notes. It generates some random values within these limits and applies them to return an input trace t_{in} from the ideal trace.

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 Section 2.2.2.

2.3.3 Computing Expected Outputs

The above method only provides input test data traces t_{in} . In order to compute the corresponding expected output traces t_{out} and carry out the test method described in Section 1, we still rely on the models of Section 2.1. and simulation techniques presented in Section 2.2.3.

2.4 Testing from an audio file

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

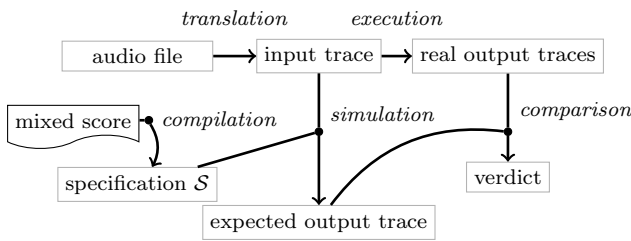


Figure 20: Audio file workflow.

case	c. miss	nb score	nb trace	time (s)
B	0	582	1843	140
	3	582	5718	334
	6	582	6387	405

Table 1: Results for the Benchmark.

in order *e.g.* to analyse a specific performance that causes errors. Such sound file can be translated into an input trace simply by marking the dates of event’s onsets. We can do that manually or using softwares, *e.g.* Antescofo itself, which can trace the events triggered when the listening machine detects them from the audio file.

As in the previous cases, we still rely on the models of Section 2.1 and simulation techniques presented in Section 2.3.3 in order to compute the expected output trace t_{out} .

3 Experiments

In this section we present the application of our MBT framework to two case studies: *B*, a benchmark made of hundreds of little mixed scores, covering many features of Antescofo’s DSL and *EIN*, a real mixed score of the piece of *Einspielung* by *Emmanuel Nunes*³.

Each case study is processed three times, with different numbers of possible consecutive missing events (0, 3 and 6 events) and a bound of 5% for the variation of the duration in the interpretation of each event.

3.1 Benchmarks

We have developed a benchmark of small scores useful to the development (debugging

³<http://brahms.ircam.fr/works/work/32409/>

and regression tests) of the system Antescofo. It aims at covering the IUT’s DSL functionality and checking the reactions of the system.

A script creates the IR and TA models, generates test suites (using CoVer), executes them according to *the first* scenario presented above (Section 1.5.1) and compares the outcome to test cases. Table 1 summarizes the results for the Benchmark *B*, reporting the number of traces generated by CoVer and the time taken by the whole test. Note that the number of traces increases with the number of missed events.

3.2 Einspielung

This second case study is a long real test case, for evaluating the scalability of our test method. It is composed of two extracts: the first 4 bars (22 events and 112 actions) and 14 bars (72 events and 389 actions) of the mixed score of *Einspielung*. Table 2 summarizes the results, with the number of IR locations, traces and testing time for each extract. CoVer did not succeed to generate the input traces for the 14 bars extract in the case of 6 possible missed events.

3.3 Feedbacks

Despite CoVer scalability problems (that can be bypassed with other scenarios), the suites of traces generated are relevant for testing the reaction of the IUT to an exhaustive set of possible performances. Some bugs in Antescofo were detected (as *e.g.* depicted in the verdict Figure 12) for specific performances, which are not easy to find manually.

We have seen that CoVer generates input test traces with minimum durations (satisfying a given reachability property), and this

case	c. miss	locations	nb trace	time (s)
EIN	0	400/1394	7/35	1/24
	3	518/1812	36/50	3/198
	6	771/2815	67/NA	97/400

Table 2: Results of experiments on *Einspielung*

may cause a problem of exponential tempo acceleration described in the example t_{in}^3 Section 1.5.2. Unfortunately it is impossible to change the choice of durations during the test input generation with CoVer.

Moreover Uppaal and its underlying model of timed automata networks also have some limitations. In particular, they cannot deal with multiple time-units, a feature of Antescofo's DSL. CoVer requires to express in extenso in \mathcal{E} all the observable messages (with $a?$ transitions), which causes an explosion in size of the environment model. The extension to patterns in the labels of transition (*e.g.* catch all symbols beginning by a) could help reducing the size of \mathcal{E} .

Conclusion and Further Work

Thanks to an ad'hoc intermediate representation for mixed scores, and a conversion into timed automata, we have developed a fully automatic offline model-based testing procedure applied to the interactive music system Antescofo, increasing the guarantee on reliability of this system. Alternative methods for the generation of test input data allow to use this test framework for different purposes, either for debugging the IMS, preparing concerts or for assistance to composition. For the latter application, we are planing an integration into the environment Ascograph for the edition of Antescofo mixed scores, with a graphical presentation of the test input and outcome.

Acknowledgments

We wish to especially thank the Uppaal's team member for their help.

References

- [1] D. A. Jaffe. Ensemble Timing in Computer Music. *ICMC*, 1983.
- [2] R. Dannenberg. Music Representation: A Position Paper. *ICMC*, 1989.
- [3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science* 126:183–235, 1994.
- [4] D. P. Anderson and R. Kuivila. A system for computer music performance. *J-TOCS* 8:56–82, 1990.
- [5] J. Arias, M. Desainte-Catherine and C. Rueda. A Framework for Composition, Verification and Real-Time Performance of Multimedia Interactive Scenarios. *ACSD*, 2015.
- [6] J. Blom, A. Hessel, B. Jonsson and P. Pettersson. Specifying and generating test cases using observer automata. *FATES*, 2004.
- [7] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella and M. Massink. Automatic verification of a lip synchronisation algorithm using Uppaal. *FMICS*, 1998.
- [8] A. Cont. A coupled duration-focused architecture for realtime music to score alignment. *IEEE TPAMI* 32(6):974–987, 2010.
- [9] A. Cont, J. Echeveste, J.-L. Giavitto, F. Jacquemard. Correct automatic accompaniment despite machine listening or human errors in antescofo. *ICMC*, 2012.
- [10] R. B. Dannenberg. Abstract time warping of compound events and signals. *Comp. Music Journal* 21(3):61–70, 1997.
- [11] J. Echeveste, A. Cont, J.-L. Giavitto, F. Jacquemard. Operational Semantics of a Domain Specific Language for Real Time Musician–Computer Interaction. *JDEDS* 23(4):343–383, 2011.
- [12] J. Echeveste. Un langage de programmation pour composer l'interaction musicale. *PhD thesis*, 2015.
- [13] H. Henkjan. From time to time: The representation of timing and tempo. *Computer Music Journal* 35(3), 2001.

- [14] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson and Skou. Testing real-time systems using Uppaal. *Formal Methods and Testing*, Springer LNCS 4949:77–117, 2008.
- [15] H. Honing. From time to time: The representation of timing and tempo. *Computer Music Journal* 25(3):50–61, 2001.
- [16] H. Honing. Structure and interpretation of rhythm and timing. *Dutch Journal of Music Theory* 7(3):227–232, 2002.
- [17] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. *SPIN*, Springer LNCS 2989:109–126, 2004.
- [18] N. Peters, T. Lossius and T. Place. An automated testing suite for computer music environments. *SMC*, 2012.
- [19] C. Poncelet and F. Jacquemard. Model Based Testing of an Interactive Music System. *ACM SAC*, 2015.
- [20] C. Poncelet and F. Jacquemard. Test Methods for Score-Based Interactive Music Systems. *ICMC SMC*, 2014.
- [21] M. Puckette. Combining event and signal processing in the max graphical programming environment. *Computer Music Journal* 15:68–77, 1991.
- [22] M. Puckette. Pure data: Recent progress. *Third Intercollege Computer Music Festival* 1–4, 1997.
- [23] R. Rowe. Interactive Music Systems: Machine Listening and Composing. *AAAI Press*, 1993.