A Framework for Composition, Verification and Real-Time Performance of Multimedia Interactive Scenarios

Jaime Arias, Myriam Desainte-Catherine, Camilo Rueda

A Framework for Composition, Verification and Real-Time Performance of Multimedia Interactive Scenarios

Jaime Arias, Myriam Desainte-Catherine
Univ. Bordeaux, LaBRI, CNRS UMR 5800
F-33400 Talence, France
email: {jarias,myriam}@labri.fr

Camilo Rueda
DECC, Pontificia Universidad Javeriana
Cali, Colombia
email: crueda@javerianacali.edu.co

***Abstract*—Interactive Scores (IS) is a formalism for composing and performing interactive multimedia scenarios. In IS, the composer defines temporal relations (TRs) between temporal objects (TOs) in order to specify the temporal organization of the scenario. During execution, the performer may trigger interaction points to modify the star/stop times of TOs, while the system guarantees that all the TRs are satisfied. IS is implemented in the tool I-SCORE and its semantics is formally defined as a Hierarchical Time Stream Petri Net (HTSPN). However, this model is not able to represent branching behaviors that are necessary to properly deal with applications such as video games and museum installations. Moreover, HTSPN does not provide tools for the automatic verification of critical properties of scenarios. In this work we define a semantics for IS based on Timed Automata (TA) and we show that such model yields to a complete framework to compose, verify and execute interactive scenarios. More precisely, we show that: 1) our model is able to deal with conditional statements in IS; 2) efficient verification techniques can be now used to reason about the written scenarios; and 3) our model allows for a directly implementation on a reconfigurable device, thus guaranteeing a real-time performance.**

***Keywords*-formal semantics; formal verification; fpgas; interactive multimedia scenarios; timed automata; uppaal**

I. Introduction

Interactive multimedia (e.g., live-performance arts and interactive museum installations) deals with the computer-based design of scenarios consisting of multimedia content that interacts with both the performer's and external actions. The multimedia content is structured in a spatial and temporal order according to the author's requirements. The potentially high complexity of these scenarios requires adequate specification languages for their complete description and verification. Moreover, the performance of some scenarios achieving compute-intensive, data-intensive or real-time tasks might not be performed properly by the standard computers and the use of supercomputers is often unfeasible due to their very high cost. Therefore, it is necessary the use of new alternatives to achieve the performance level needed for the execution of interactive multimedia scenarios.

Interactive Scores (IS) [1], [2] is a formalism for composing and performing interactive multimedia scenarios. In IS, the performer has the possibility to influence the execution of scenarios by triggering interaction points (IP). Hence, the performer enjoys a certain freedom in choosing the time of interaction (or whether it takes place) leaving the system the task of maintaining the temporal constraints defined by the composer. Scenarios are composed of textures and structures. Textures represent the execution in time of multimedia processes. Structures allow to design modular scenarios and impose a hierarchical organization on them. The temporal organization of the above temporal objects (TOs) is defined by asserting temporal relations (TRs) those objects should obey. The IS model combines two temporal paradigms used in current multimedia tools [2]: *time-line* and *time-flow*. The former is represented at composition time when the composer defines multimedia processes by their start and end times, as well as by TRs between them. The latter is represented by the time at which the processes are actually executed.

Currently, the IS model is implemented in I-SCORE (www.i-score.org), a tool that offers two different stages: *composition* and *performance*. In the former, composers place TOs on a horizontal time-line. Then, they add IPs and connect TRs between the TOs in order to define the temporal properties of the scenario. During the performance stage, the performer can dynamically trigger the IPs while the system maintains the temporal properties defined by the composer (i.e., the TRs). In I-SCORE, the scenarios are executed by an abstract machine, called *ECO machine*, that relies on a Hierarchical Time Stream Petri Net (HTSPN) [3] to represent and execute the partially ordered set of events [4]. Thus, each time a scenario is written or modified, it must be translated into a HTSPN to be executed.

During the last few years, I-SCORE has been used in applications such as video games, live-performance arts, and interactive museum installations. However, these kind of applications demand flexible control structures (i.e., conditionals and loops) which are not supported in the current formal model of IS. Several researchers have made many efforts to extend IS with control structures (e.g., process calculi, event structures, colored petri nets), but there is no practical solution for the automatic verification and real-time performance of scenarios [5]–[7].

In this paper, we present a timed automata (TA) [8] based framework to address the challenges in the modelling, verification and real-time performance of multimedia interactive scenarios. In the proposed framework, we model scenarios as a network of TA. Moreover, we add conditions to IPs allowing the specification of branching behaviors in IS. We take advantage of the mature and efficient tools for TA to simulate and verify automatically scenarios. Furthermore, we implemented a tool to construct TA models automatically from the intuitive composition environment of I-SCORE. Once the scenario satisfies the author's requirements, the verified TA model is synthesized into a reconfigurable hardware (i.e., FPGAS) in order to guarantee its real-time execution.

The main contributions of our framework are: (1) a novel model for the specification of interactive scenarios with non-linear behaviors; (2) an automatic tool to construct bottom-up TA models from scenarios written in I-SCORE, allowing a friendly environment for composing; (3) an automatic verification of scenarios using mature and efficient algorithms of symbolic model checking (i.e., UPPAAL); and (4) a real-time and low-latency performance of scenarios by executing the verified model on FPGAS.

The rest of the paper is organized as follows. We start by presenting the IS formalism in Section II. Also, we briefly introduce the TA model and the tool UPPAAL. In Section III, we develop the TA model for IS endowed with conditionals. Next, in Section IV we introduce our tool to automatically construct a TA model from I-SCORE. Moreover, we show the verification of some critical properties of scenarios using UPPAAL and we discuss their real-time performance on FPGAS. We conclude in Section V by pointing out to related work and discussing on some ideas for future work.

II. PRELIMINARIES

A. Interactive Multimedia Scenarios

Interactive Scores (IS) [1], [2] is a formalism for composing and performing interactive multimedia scenarios (e.g., live-performance arts) where the performer has the possibility to influence their execution. Roughly, the composer partially defines the temporal organization of the scenario by means of temporal relations (TRs) between temporal objects (TOs). In IS, the performer can modify, during performance, the starting and the stopping of TOs by triggering interaction points (IPs) while the system maintains the temporal constrains defined by the composer. Therefore, a scenario could have a set of possible interpretations that share the same temporal properties.

As we mentioned above, composers partially define the temporal organization of their scenarios by adding TRs between TOs. TOs are classified into *textures* and *structures*. Textures represent the execution in time of a given multimedia process (e.g., a process changing the brightness of a lamp). Structures allow to design modular scenarios (i.e., a hierarchical organization) and denote only the execution of a group of TOs with their own temporal organization. In this regard, a scenario can be seen as a structure that contains the temporal organization of the TOs placed by the composer. Each TO has associated a set of control points that represent particular moments of its execution, for example, the start and the end. The possibilities of interaction are expressed by means of IPs that turn a control point into a dynamic one. Therefore, a dynamic control point must be explicitly triggered by the performer during the execution while the other control points (i.e., the static control points) are triggered by the system.

TRs define two qualitative relations between two control points: *precedence* and *posteriority*. These relations are taken from point algebra and they are symmetrical. Moreover, TRs are enhanced with quantitative constraints by giving a range of possible durations in $[0, \infty]$. Thus, the composer must define a minimum duration (Δ_{min}) and a maximum duration (Δ_{max}) for each TR. Depending on the above values, TRs can be classified as: (1) *rigid*, if $\Delta_{min} = \Delta_{max} > 0$; (2) *synchronization*, if $\Delta_{min} = \Delta_{max} = 0$; (3) *flexible* or *supple*, if $\Delta_{min} = 0$ and $\Delta_{max} = \infty$; and (4) *semi-flexible* or *semi-rigid*, if $\Delta_{max} \neq \infty$ and $\Delta_{min} \neq \Delta_{max}$. For the sake of presentation, we shall use the notation $[x, y]$ to denote that the minimum and the maximum duration of a TR are x and y, respectively.

Currently, the IS model is implemented in the software called I-SCORE. This software provides two different stages: *composition* and *performance*. During composition stage, composers place TOs, represented as *boxes*, on a horizontal time-line. Then, they add IPs and connect TRs between the TOs in order to define the temporal properties of the scenario. Fig. 1 shows a scenario designed in I-SCORE that contains seven boxes: `A`, `B`, `F` and `G` are textures whereas `C` and `D` are structures. We recall that the whole scenario is also a structure. Lines between TOs represent TRs and lines over TOs express their duration. Dotted lines symbolize the interval of time in which the performer can trigger the corresponding IP. Flags over boxes represent IPs. In I-SCORE, a box with no explicit TRs defining its start time, imposes an implicit TR with the starting of its parent (e.g., box `F`). In the case in which a box has an IP at the start (e.g., box `A`), the added TR is flexible (i.e., its duration is not bounded). For instance, the starting of boxes `A` and `B` is define by an implicit TR between the starting of the scenario and them.

Since during composition stage the computation time is not critic, the scenario is viewed as a Constraint Satisfaction Problem (CSP). Thus, when the composer changes the characteristics of a TO (i.e., the start time and duration), a constraint solver propagates the new constraints, which leads the TOs to automatically move or stretch in order to keep the temporal properties imposed by the composer.

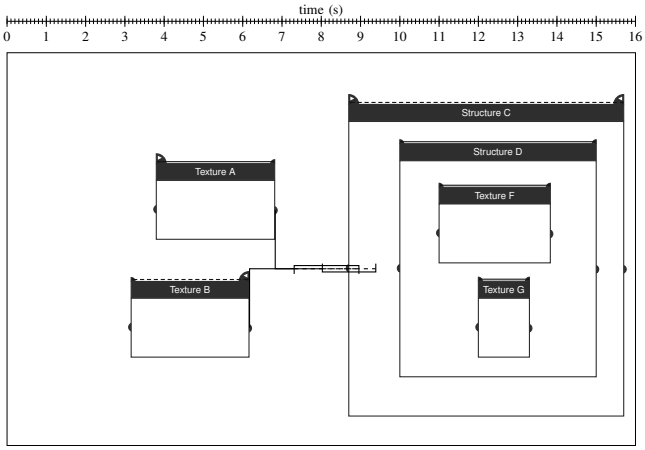

Figure 1. Example of an interactive scenario. Boxes A, B, F and G are textures, whereas C and D are structures.

On the other side, during performance the performer can dynamically trigger the IPs while the static control points are triggered by the system. In I-SCORE, multimedia processes are executed by external applications such as MAX/MSP (http://cycling74.com) or PURE DATA (http://puredata.info). Then, it uses multimedia protocols like OSC (http://opensoundcontrol.org/introduction-osc) to send the messages defined by the composer to specific applications. IPs are also triggered by specific messages that are asynchronously sent by the performer during performance. It is important to note that the system will refuse the triggering of an IP if its message is sent outside of the interval of time defined by the composer. However, the system will automatically trigger the IP when the maximum duration of the interval has elapsed and it has not been triggered yet. In this way, the system maintains the temporal properties imposed by the composer. Now, let us show an example of an interactive scenario to better understand the notions that we have introduced so far.

Example 1. Assume that the scenario in Fig. 1 specifies a fragment of a theatrical installation. The scenario aims to reproduce the atmosphere of a cloudy and dark forest. Imagine that texture B controls a machine that produces white smoke in order to create the cloudy atmosphere. Once the amount of smoke is enough, the performer can trigger the IP of B to stop the machine. As constraints, B must start at 3160 ms and stop only when the performer decides it (i.e., its duration is not bounded). On the other hand, the aim of the texture A is to spread the smoke over the scene by controlling a collection of fans. The performer can start A, by triggering its IP, from the beginning of the act (i.e., the starting time of A is not bounded). Moreover, the composer knows that 3014 ms are necessary to obtain the desired result (i.e., the duration of A). Once the cloudy atmosphere is recreated, the howl of a wolf (i.e., texture F) sounds for 2832 ms, and while this happens, a beam of a yellow light (i.e., texture G) pierces the cloudy forest during 1184 ms giving the impression that a car is approaching. The composer knows that the effect created by the smoke and fans lasts a time before it disappears. Then, he/she "encapsulates" F and G within the structure C and adds two TRs to ensure that the content of C is executed after the atmosphere is well created and before it disappears. The first TR is between A and C and its duration is [1200, 2560] ms. The second TR is between B and C and its duration is [1136, 2784] ms. The performer can only start C within the interval of time in which these TRs are satisfied.

In order to execute the written scenarios, an abstract machine, called *ECO machine*, is used [4]. This machine is responsible of: (1) triggering the static control points (i.e., the starting/stopping of TOs with no IPs); (2) controlling, in real-time, the triggering of the dynamic control points (i.e., IPs); and (3) maintaining the temporal organization of the scenario. The operation of the machine is described in terms of state transitions that are synchronized with a global clock. This machine relies on a Hierarchical Time Stream Petri Net (HTSPN) [3] to represent and execute the partially ordered set of events. Therefore, each time a scenario is written or modified, it must be translated into a HTSPN to be executed. The method to construct the HTSPN model from a scenario is the following [4]: each control point is turned into a transition. If a temporal constraint imposes the simultaneity of different control points, their transitions are merged. If a precedence relation is specified between two control points, a sequence arc/place/arc is added between the transitions that represent them. The range of time that represents the duration of a TR is defined over each arc and it represents the possible durations of the relation. Furthermore, the firing of a transition that represents a dynamic control point is conditioned by receiving an external control message. It is important to note that each ingoing arc of this kind of transition has minimum and maximum duration values that correspond with the range of time decided by the compositor. The minimum value corresponds to the minimum time at which the transition can be crossed whereas the maximum value corresponds to the time at which the transition will be automatically crossed.

B. Timed Automata and UPPAAL

Timed Automata (TA) [8] is a formalism for modelling and verification of real-time systems. Intuitively, a timed automaton is a finite-state machine extended with non-negative real-valued variables that model the logical *clocks* of the system. Clocks are initialized with zero when the system is started, and then increase synchronously with the same rate. The behavior of the automaton is restricted by imposing clock constraints on *transitions* (i.e., edges). Thus, a transition can be taken when the clocks values satisfy its

guard. An associated *action* is executed when a transition is taken. Moreover, a set of clocks may be reset. *Locations* (i.e., states) have invariants that are predicates over clocks. Thus, a timed automaton can stay at a state as long as its invariant is satisfied.

Next, we present the formal definition of timed automata [8]. Let C be a finite set of real-valued variables denoting the set of all clocks, and $\mathcal{B}(C)$ the set of conjunctive formulas of atomic clock constraints of the form $x \bowtie n$ or $x - y \bowtie n$ where, $n \in \mathbb{N}_0$, $x, y \in C$ and $\bowtie \in \{\leq, <, =, >, \geq\}$.

Definition 1 (Timed Automaton)**.** A timed automaton $\mathcal{A}$ is a tuple $\langle L, l_0, \Sigma, C, E, I\rangle$ where L is a finite set of locations, $l_0 \in L$ is the initial location, Σ is a finite alphabet denoting actions, C is a finite set of clocks, $E \in L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$ is the set of edges between locations, and $I : L \rightarrow \mathcal{B}(C)$ assigns invariants to locations.

UPPAAL [9] is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with features to facilitate the modelling tasks, such as bounded global/local integer variables, structured data types, channel synchronization, committed/urgent locations and priorities. A *network of* TA is the parallel composition $A_1 | \ldots | A_n$ of a set of TA $A_1, \ldots, A_n$, called *processes*, combined into a single system by the CCS parallel composition operator [10] with all external actions hidden. On one hand, synchronous communication between automata is carried out by hand-shake synchronization using input and output actions. To model hand-shake, the action alphabet of TA is assumed to consist of symbols for input actions denoted as $a?$, output actions denoted as $a!$, and internal actions represented by the distinct symbol τ. On the other hand, asynchronous communication is carried out by shared variables.

As said before, UPPAAL considers integer bounded *variables*. Predicates over them can be used as guards on the edges and their values can be updated using resets on the edges. In our model, global variables are used to represent the asynchronous message passing between the environment and the system. Variables are highly necessary to model conditionals and multimedia processes in IS. Additionally, UPPAAL allows to model atomic sequences of actions using *committed locations* where no delay is allowed. That means that, if any process is in a committed location then only action transitions starting from such a committed location are allowed. Moreover, time is not allowed to pass when the system is in an *urgent* location. Finally, *broadcast channels* allow one process to synchronize with multiple processes.

III. Timed Automata Semantics for IS

In this section we present a formal semantics of IS using TA. Our model follows the modelling patterns described in [11] in order to design a clear and structured model.

Intuitively, a scenario can be seen as several processes (i.e., TOs and TRs) running in parallel and whose starting and stopping dates depend on the behavior of the others. For instance, a process controlling the brightness of a lamp (texture `L`) starts five seconds after the stopping (relation `r`) of a process playing a sound (texture `s`). Hence, we can model a scenario as a *network of TA* in which both TOs and TRs are modelled as TA processes. The starting and stopping of each timed automaton (i.e., the temporal organization of the scenario) is defined by its synchronization with the others. We use *broadcast channels* for the communication since a timed automaton can synchronize with more than one simultaneously.

Let us now present in more detail our TA model by defining the timed automaton of a TR and a TO. We shall also present the timed automaton modelling an IP and a mechanism to handle the temporal constraints imposed by TRs. We shall see that our model allows to represent conditionals that are not supported by the current HTSPN model of IS. Moreover, we can automatically verify some essential properties of scenarios using the model checker provided by UPPAAL.

A. Modelling Temporal Relations

As we explained before, a TR can be classified depending on its duration (see Section II-A). Intuitively, a TR whose duration is completely defined (i.e., a *rigid* relation) can be seen as a simple delay between two TOs. A TR whose duration is partly defined (i.e., a *semi-flexible* or *flexible* relation) can be seen as a delay whose duration is an interval of time (i.e., it has a maximum and a minimum duration). Next, we present the TA model for the above types of TR. A model for the *synchronization* relation is not necessary because we can synchronize the starting/stopping of two TOs by means of events.

Rigid temporal relations. A rigid TR is defined as the timed automaton in Fig. 2. It begins in the state `idle` and remains on it until the event `event_s` is emitted. This event starts the execution of the TR. Once this occurs, the timed automaton stays in the state `wait` until the duration `dur` elapses. Observe that this state models the delay generated by the TR. Once the delay finishes, the timed automaton moves to the state `finished` and emits the events `event_e1` and `event_e2` representing the elapsing of the minimum and maximum duration of the TR, respectively. These events may define the starting or the stopping of other TA.

Let us explain the remaining states of the above timed automaton through an example. Imagine that a TR is inside of a structure that is stopped by an IP while the TR is executing. In that case, the TR is killed (we use the term *kill* to denote the sudden stopping of a TR as a result of the stopping of its parent) by the event `kill_p` that is emitted by its parent (i.e., the stopped structure). Moreover, the timed

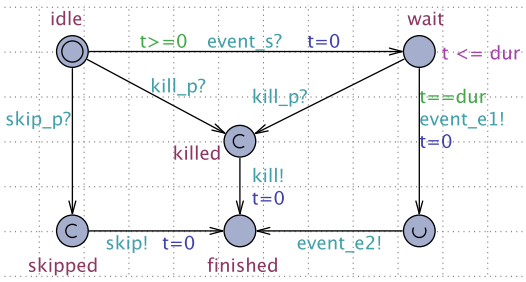

Figure 2. Timed automaton modelling a rigid TR.

automaton emits, at the same time, the event `kill` in order to stop other TA. Later, we shall introduce the events `skip_p` and `skip`.

Flexible and semi-flexible temporal relations. We present in Fig. 3 the timed automaton modelling both a flexible and a semi-flexible TR. We recall that the difference between these two types of TR is that the maximum duration of a flexible TR is not bounded (i.e., infinity).

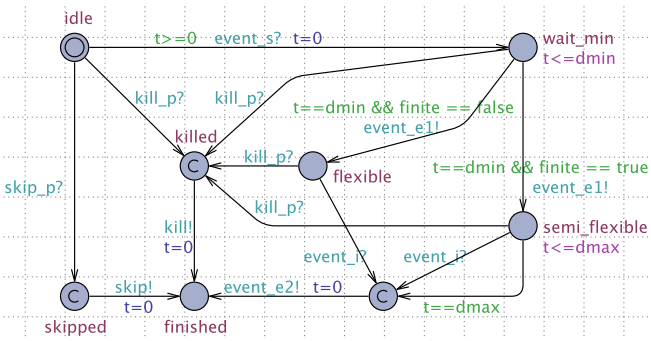

Figure 3. Timed automaton modelling a flexible and a semi-flexible TR.

Similar to the above model, the timed automaton starts in the state `idle` and moves to the state `wait_min` when the event `event_s` is emitted. Then, it stays in the state `wait_min` until the minimum duration (`dmin`) elapses. Once this occurs, the timed automaton emits the event `event_e1` and goes to the state `flexible` if the maximum duration (`dmax`) is infinity, otherwise it goes to the state `semi_flexible`. The event `event_e1` synchronizes with TA that are waiting for the elapsing of the minimum duration of the TR.

In the case of a semi-flexible TR, the timed automaton waits for either the elapsing of the maximum duration or the emitting of the event `event_i` to stop the TR (state `semi_flexible`). In the case of a flexible TR, it only waits for the event `event_i` to stop (state `flexible`). The event `event_e2`, that represents the stopping of the TR, is emitted once the TR finishes. As explained below, the event `event_i` represents either the triggering of an IP or the stopping of other TR. The remaining events and states of the timed automaton specify the same behavior explained in the TA model of a rigid TR.

Handling temporal relations. Composers usually define the starting time of TOs by means of one or more TRs. For instance, in our running example (scenario in Fig. 1) the starting of structure `C` is defined by two semi-flexible TRs. Before defining the TA to handle this complex behavior, let us present an operational intuition of using two or more TRs to define a temporal constraint.

Roughly, all TRs are held when the minimum duration of all TRs have already elapsed and no TR has reached its maximum duration. Assume that the two TRs defining the starting of a TO are those presented in Fig. 4. Hence, the TO can start between 10 and 13 s by triggering an IP. It is important to note that in IS, the TO will start automatically at 13 s if the IP is not triggered within the above interval of time.

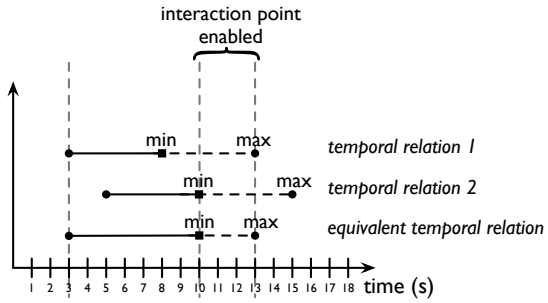

Figure 4. Operational intuition of two TRs. They are held when all TRs have reached their minimum duration and no TR has reached its maximum duration.

We can now introduce the timed automaton (see Fig. 5) for handling complex temporal properties defined by means of n number of TRs. The timed automaton starts in the state `idle` and waits for either the elapsing of the minimum duration (event `event_s1`) or the stopping (event `event_s2`) of a TR. The timed automaton stops (state `finished`) once all TRs have reached their minimum duration (i.e., `counter = n`). Moreover, it emits the event `event_e` allowing the synchronization with other TA modelling, for example, an IP or the starting of a TO. Nevertheless, an error is produced (state `error`) when a TR stops before all TRs have reached their minimum duration (i.e., the temporal property defined by the composer cannot be satisfied). The local variables `counter` and `skip_v` are initialized with values 0 and `true`, respectively. The event `kill_p` models the same behavior as we have already explained.

Interaction points. Roughly, IPs are events asynchronously triggered by the performer during the execution of the scenario while the TRs are maintained by the system. In our model, we take advantage of the shared variables supported by UPPAAL to model the asynchronous communication between the user and the scenario. Moreover, they will allow us to enhance the current model of IS with *conditionals*.

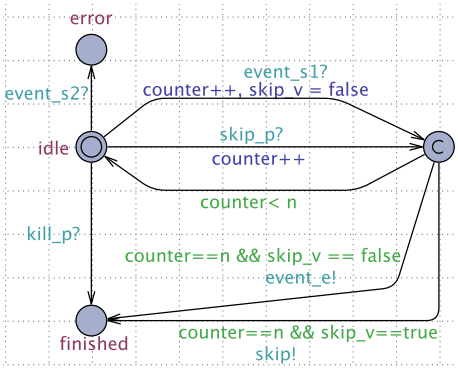

Figure 5. Timed automaton for handling TRs.

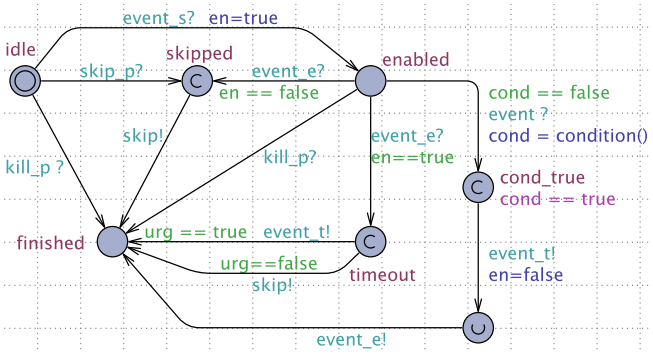

Figure 6. Timed automaton modelling a conditioned IP that allows to describe branching behavior in IS.

In the proposed model, we extend IPs with conditions (conditioned IPs). Let us explain this notion through the following example. Assume that the IP for starting the structure `C` of our running example can only be triggered if the sending event (e.g., the temperature of the environment) has a value greater than $30°C$. However, if this event is not sent or its value does not satisfy the condition during the interval of time defined by the composer, he or she must define whether the IP must be triggered automatically (*urgent behavior*) or not. The urgent behavior corresponds to the normal execution of IPs in the current model of IS. In the second case, the execution of the following TOs and TRs (i.e., the *branch*) will be omitted. For example, if the texture `A` of our running example has a conditioned IP at the start and it is not triggered, then the texture `A` and its following TR with structure `C` will not be executed. Hence, the starting of structure `C` is only defined by the TR with the texture `B`.

Now we are ready to present the timed automaton modelling a conditioned IP (see Fig. 6). The timed automaton begins in the state `idle` and waits for the event `event_s` to start listening the events sent by the performer (state `enabled`). The event `event_s` is emitted when all TRs have reached their minimum duration. The automaton remains listening for the event until either (1) the IP is enabled (explained above) and the event `event_e` is emitted (state `timeout`) or (2) its value satisfies the condition (state `cond_true`). We defined the function `condition()` to verify whether the value of the sent message satisfies or not the condition of the IP. The event `event_e` is emitted when the maximum duration of a TR elapses. Note that the case (2) represents the scenario in which the IP is not triggered or its value does not satisfy the condition within the valid interval of time. Thus, depending on the behavior defined by the composer (parameter `urg`), the IP will be triggered automatically (i.e., event `event_t`) or the execution of the branch will be omitted (i.e., event `skip`). In the case (1), the IP is triggered (i.e., event `event_t`) since the condition is satisfied. Moreover, the event `event_e` is emitted in order to stop the TRs that are still waiting for the elapsing of their maximum duration. Once an IP is triggered, the TRs defining the temporal property of the corresponding TO are no more relevant during the performance of the scenario.

The event `kill_p` specifies the same behavior as we have already explained. The remaining states are introduced through the following example. Assume that two new textures `A` and `B` control the playing of two different videos whose performing depends on the room lightning (i.e., light or dark). Thus, each texture has a conditioned IP listening the same event during the same interval of time. However, the defined conditions are mutually exclusive thus only one IP will be triggered and the execution of the other branch will be omitted. In this regard, we use the shared variable `en` as a global flag that allows to stop the IPs whose condition is not satisfied (state `skipped`) once the other IPs listening for the same event have been triggered and have also changed the value of `en` to `false` (i.e., their conditions hold).

As we explain before, the omission of the execution of a branch causes that all TRs and TOs of the branch are skipped. For this reason, each timed automaton of our approach models the above behavior by leaving out its execution when the event `skip_p` is emitted. Moreover, the skipping of the branch is propagated by sending the event `skip`. For instance, in the above example the IP whose condition is not satisfied will omit the execution of the corresponding texture and its branch.

Performer interaction. IPs allow users to interact with the scenario during performance. This interaction is carried out by asynchronously sending messages to the system. We model this non-deterministic environment using the timed automaton in Fig. 7. Roughly, it emits the event `event` (i.e., the message) with an attached value (`val`). The value is communicated to the IPs by means of the shared variable `msg`. The event is emitted at a non-deterministic time (i.e., the transition is not guarded by clock constraints) and it is synchronized with the IPs that are waiting for it. Several copies of this automaton could be instantiate in order to represent different user interactions.

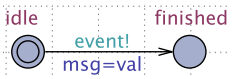

Figure 7. Timed automaton modelling an user interaction.

B. Temporal Objects

In the following, we introduce the models to describe textures and structures. As we shall see, TOs can be represented using the automata defined above. That allows to have a simple, yet powerful, modular model of IS.

Textures and multimedia processes. As we saw in Section II-A, in IS a texture is the same as a TR, but the former has an attached multimedia process that is executed in time by an external application. Therefore, a texture with an IP at the end (e.g., texture `B` in the running example) can be modelled using the timed automaton of a flexible or semi-flexible TR. Otherwise, it is modelled as a rigid TR.

Unlike the current model of IS, in our approach a texture can have one or more attached multimedia processes, thus decreasing the number of textures executed concurrently (i.e., a reduction in the size of the scenario). Roughly, a multimedia process is modelled as a list of parameters (i.e., values that are sent to the external application) associated with their synchronization time. Let us explain the above with an example. Assume that the texture `F` of our running example has attached the multimedia process in Fig. 8. Observe that it is composed of seven parameters (i.e., the points p_i) that are sent at Δ_i time after the above (i.e., *intra-stream synchronization*) in order to control in time the brightness of a lamp.

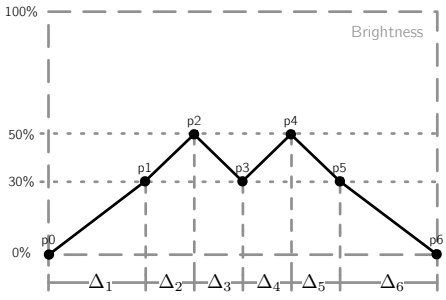

Figure 8. Example of a multimedia process controlling the brightness of a lamp.

Before formalizing the idea, we show how to represent the parameters of a multimedia process. UPPAAL allows to create new data structures as in the C language. We thus implement a structure (`parameter_t`) composed of the value and the synchronization time of the parameter. Hence, a multimedia process is a list of `parameter_t` elements. Fig. 9 shows the implementation of the example presented above.

```
typedef struct {
  int value;
  int offset;
  } parameter_t;

parameter_t process_brightness[7] = {
  {0,0}, {30,5}, {50,2}, ... };
```

Figure 9. Data structure representing the parameters of a multimedia process. The variable `process_brightness` corresponds to the process in Fig. 8.

Now we are ready to present the TA model for a multimedia process (see Fig. 10). The timed automaton begins in the state `idle` and its beginning is synchronized with the starting of a specific texture (event `start`). Once this occurs, the parameters of the multimedia process begin to be sent maintaining their synchronization time (i.e., state `wait`). The event `send` represents the communication with the external application and the shared variable `data` denotes the sending value. The timed automaton stops either the texture has stopped (event `stop`) or all parameters have already been sent (i.e., `index = limit`). The events `kill_p` and `skip_p` model the same behavior as we have already explained.

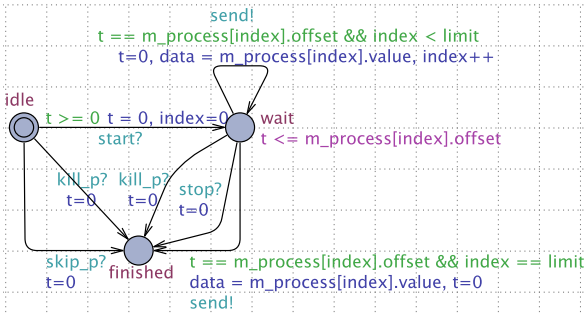

Figure 10. Timed automaton modelling a multimedia process.

Structures. Intuitively, a structure defines the temporal organization of a group of TOs. For example, the structure `D` in the running example defines the starting of textures `F` and `G`. Moreover, the stopping of `D` causes the stopping of its children regardless of whether they are running. It is important to note that TRs can only be defined between TOs in the same hierarchy level (i.e., scope).

In the spirit of textures, we can model structures as flexible or semi-flexible TRs with an attached group of TOs (children). Roughly, a structure with an IP at the end is modelled as a flexible or semi-flexible TR. Since the stopping of a structure causes the stopping of its children, we use the timed automaton in Fig. 11 to emit a kill event (i.e., event `event_out`) when the structure emits its stopping event

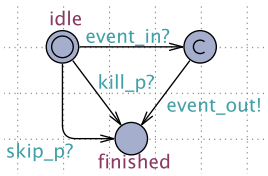

Figure 11. Auxiliary timed automaton to stop the children of a structure.

(i.e., event `event_e2` in the timed automaton of a TR). This event will synchronize with the `kill_p` of its children leading them to stop. Notice that the above behavior is propagated down the hierarchy stopping all descendants of the structure.

In the other case, a structure with no IPs at the end is modelled as a flexible TR whose minimum duration is the duration of the structure and whose maximum duration is infinity. These considerations are necessary since the structure must wait for the duration imposed by the composer and also for the ending of all its children. We use the timed automaton to handle TRs (defined above) in order to know when all its children have stopped. Therefore, this timed automaton will emit the event to stop the structure (i.e., event `event_e`) when both the minimum duration of the structure elapses (i.e., event `event_e1` in the TR model) and its children have stopped (i.e., event `event_e2` in the TR model). Note that the event `event_e` has the same effect as an IP because it stops dynamically the TR (i.e., the structure) during its execution.

Interactive multimedia scenarios. As a result of the above, we can conclude that an interactive multimedia scenario is a network of TA representing the execution in parallel of the TOs and TRs defined by the composer in the scenario. Thus, as we saw before, the whole scenario is modelled as a structure with an IP at the start and containing these elements, and whose starting time is defined by an event sent by the performer (e.g., the play button).

IV. Verification of Interactive Scenarios and their Real-Time Performance

In this section we shall describe how to translate scenarios written in the software I-SCORE into our TA model in order to verify some critical properties of them. Moreover, we present a hardware implementation of the verified scenarios that allows their real-time execution. The reader can found the details of the implementations presented below as well as examples at http://www.labri.fr/perso/jarias/is-framework.

A. Verification of Interactive Scenarios Using UPPAAL

As we saw in Section II-A, I-SCORE provides a graphical environment for composing and performing interactive scenarios. Its current implementation (version 0.3) allows to write conditions on IPs, but it still lacks a formal modelling. We thus implemented a tool in OCAML (http://www.ocaml.org) to automatically construct UPPAAL models from scenarios written in I-SCORE. In this way, we take advantage of: (1) the composition stage of I-SCORE to intuitively compose scenarios; and (2) the mature and efficient model checker provided by UPPAAL to verify the properties of scenarios. Roughly, our tool parses the XML file of the scenario generated by I-SCORE and applies the definitions described in the above section in order to create an XML file (supported by UPPAAL) of the generated bottom-up TA model of the scenario. Since a scenario contains a finite number of TOs and TRs our tool always terminates.

Once the TA model is built, we can use UPPAAL to automatically verify properties of scenarios. Let us now show the verification of some properties of the scenario in Fig. 1.

Terminating scenarios. In I-SCORE, adding an IP at the end of a TO causes that the duration of the TO is not bounded (i.e., the maximum duration is infinity). Moreover, a TO with an IP at its start (e.g., texture `A` in the running example) implicitly defines a flexible TR from the start of its parent and itself. By checking with UPPAAL

```
A<> Scenario.finished      (Property is not satisfied)
```

we can prove that the above assumption of I-SCORE produces a scenario that may never terminate. A simple execution in which this property is not satisfied is when no IP is triggered during the performance. We can solve this problem by bounding the duration of (1) the TR preceding the texture `A` and the duration of the TOs (2) `B` and (3) `C`. For instance, we changed the duration of (1), (2), and (3) to [0,3776] ms, [0,3120] ms and [0,10000] ms, respectively. However, the property remains being not satisfied. We show the cause of this by checking the property below.

Playability. This property is very important in IS because a scenario could be over-constrained and therefore not playable. Considering the previous changes, the TRs defining the starting time of structure `C` may not be always satisfied. By proving

```
A[] !Control_Start_C.error
                           (Property is not satisfied)
```

where `Control_Start_C` is the timed automaton handling the TRs of `C`, we can see that the TRs are not always satisfied (i.e., the timed automaton reaches the state `error`). An execution in which this happens is when the texture `A` starts at 144 ms and stops at 3168 ms, and texture `B` starts at 3168 ms and stops at 6288 ms (i.e., the IP is not triggered). However, if we bound the duration, respectively, of texture `B` and its TR to `C` to [1000,2000] ms and [300,4000] ms, we can prove that the TRs of the scenario are always maintained. Moreover, the above modification makes the termination property (described above) becomes satisfied.

Temporal properties of TOs. As we explained before, composers use TRs to define temporal constraints on the starting and stopping times of TOs. In I-SCORE, it can sometimes be complicated to know the result of adding

many TRs since the system does not provide a feedback of the resulting temporal constraint. However, we can use UPPAAL to check that a desired constraint will always be satisfied. For instance, assume that the composer hopes that the structure C always starts after 4468 ms. We can prove that this property is always satisfied by verifying that

```
E<> Structure_C.wait_min && clk < 4468
                          (Property is not satisfied)
```

where the variable clk denotes the current time of execution of the scenario. Intuitively, we ask if there exists an execution of the scenario in which the structure C starts before 4468 ms.

Shared resources. We recall that textures use external applications to execute in time the multimedia processes. Hence, a resource (i.e., an external application) cannot be used by two or more textures at the same time. For instance, assume that textures F and G, in the running example, send values to the same external application which controls the playing of a video. If both textures send values at the same time, the application may not be able to handle the data or the expected behavior could not occur. We can prevent such behavior by proving the following property:

```
E<> Texture_G.wait && Texture_F.wait
                              (Property is satisfied)
```

We then see that both textures can eventually be executing simultaneously. We can solve this problem, for example, by modifying the duration of the texture G and its preceding TR by 1000 ms and 100 ms, respectively. In that case, the property is not satisfied and both textures will never be executed simultaneously. Fig. 12 shows the results of verifying the above properties once the suggested modifications are applied to the scenario. Observe that now the scenario satisfies the expected properties.

B. True Parallel Implementation of IS

A Field Programmable Gate Array (FPGA) is a digital circuit that can be reprogrammed, many times, to the desired functionality requirements of the user after manufacturing.

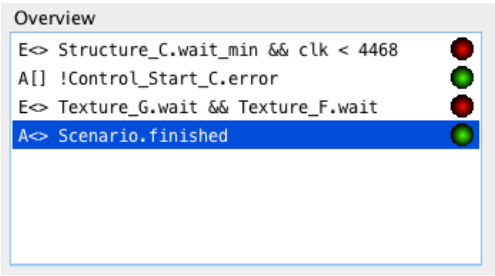

Figure 12. Overview of the verification of some properties of the scenario once the suggested modifications are applied. The green light means that the property is satisfied, while the red light means the opposite.

FPGAs can simultaneously compute millions of operations in resources distributed on the device (i.e., spatial computing). Then, such systems can be hundred of time faster than microprocessors-based systems and also provide huge power, area, and performance benefits over software. In recent years, FPGAs have been used with success in many different areas of application, such as aerospace, automotive, medical, video and audio processing applications [12]–[14]. Since FPGAs have risen over the last years, they have become economically viable for use in several applications.

Let us now present the benefits of these devices [15]:

- *Reconfigurability:* FPGAs can be reconfigured at any time.
- *High-level design:* The hardware is defined by using high-level hardware description languages. Moreover, the designed systems can be simulated and verified before their execution on the FPGA.
- *Physical parallelism:* FPGAs allow to design completely parallel systems without computation loading.
- *High-speed:* Parallelism and fast clock rates of FPGAs allow systems to achieve very high speed that sometimes outperforms processor-based systems.
- *Reliability:* FPGAs provide true hardware reliability because there is no operating system or driver layer that can affect system update.
- *IP protection and re-use*: It is difficult to reverse engineering a synthesized system. Moreover, a tested hardware design can be re-used multiple times by instantiating. As we shall see, we construct bottom-up scenarios using the tested modules of each timed automaton presented above.

The creation of a FPGA-based system consists on building a bitstream file to load into the device. The designers start with an application written in a Hardware Description Language (HDL), such as SystemVerilog or VHDL. This abstract design is optimized to fit into the FPGA's available logic. Next, the optimized design is mapped into logic blocks and routing determines the interconnected resources. Finally, the bitstream file is generated in order to configure the logic blocks and routing resources of the FPGA appropriately. Once the bitstream file is loaded into the FPGA, it operates as a custom digital system.

In our framework, the verified scenarios can be synthesized into a reconfigurable hardware (i.e., an FPGA). Thus, we provide a real-time and low-latency performance of scenarios, which is not guaranteed in I-score, by taking advantage of all the benefits listed above. Moreover, FPGAs are synchronous hardware with a jitter less than one cycle of clock and they are not affected by the complex behavior of the operating system services, interrupt handling, etc.

As we explained before, a timed automaton is a finite-state machine (FSM) extended with non-negative variables that model the clocks of the system. Since the verification of TA is an integer based formalism, then the use of integer

variables in our implementation does not affect the behavior of the modelled scenario. Channels can be implemented as wires connected between each process (i.e., timed automaton) with a logic to handle multiple connections.

Let us now present the proposed hardware implementation of a timed automaton (see Fig. 13). A Mealy FSM was chosen to model a timed automaton because it adequately expresses the behavior of synchronous systems: (1) the outputs depend on the current state and the inputs, and (2) the outputs react instantaneously to the inputs. Moreover, we used SYSTEMVERILOG (SV) as HDL to describe our implementation. Roughly, we implemented parametric templates for each timed automaton presented in Section III. The specification of our model is natural in SV since it combines the features of HDL such as VERILOG and VHDL with features from specialized Hardware Verification Languages (HVL), together with features from C and C++.

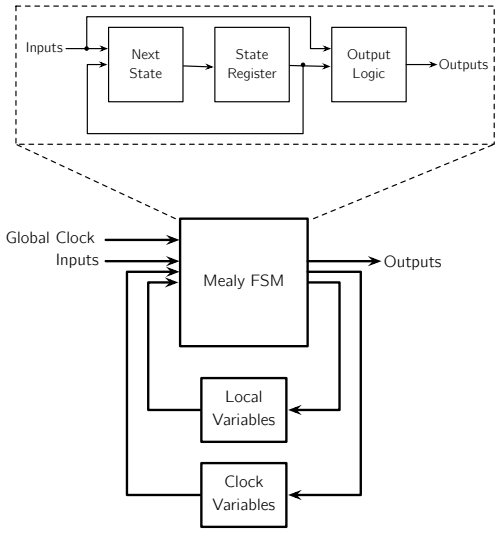

Figure 13. Block diagram of the hardware implementation of a timed automaton.

In our approach, we generate the clock of the scenario from the FPGA clock. For that, we divide the frequency of the FPGA clock by the number of clock cycles needed to obtain the desired frequency. On the other side, the FPGA clock is used for the FSM. That is very important because we need to guarantee that the sampling period of the scenario is greater than the time needed to update the states of the FSMS.

Each time that a timed automaton resets its local clocks, it updates the register `Clock Variables` with the global time of the system (i.e., the input `Global Clock`). Thus, to calculate the elapsed time of each local clock, the FSM subtracts the stored value to the global clock. In this way, the system is synchronized with the same clock rate. On the other side, the local variables of the timed automaton are stored in the register `Local Variables`. As we can see, our model does not need special architecture for its implementation. The results of the simulations show that our approach allows to satisfy all real-time constraints imposed by the composer and also it provides a low-rate data synchronization that allows to satisfactory react to the environment's events.

With the help of the XILINX VIVADO DESIGN SUITE (http://www.xilinx.com), we can simulate the generated SV code. We also can validate the generated code with the result of the UPPAAL simulation by implementing test benches. We have done large amounts of behavioral comparisons to validate the consistency between the TA model and the generated code. Finally, the generated code can directly be synthesized into an FPGA to its execution. It is important to note that the only limitation of our implementation is the number of logic blocks provided by the FPGA platform.

V. CONCLUDING REMARKS AND RELATED WORK

In this paper, we have presented a novel framework for composition, verification and real-time performance of interactive multimedia scenarios. We defined the semantics of IS using a network of TA and we extended it with the ability to express branching behaviors. Moreover, we implemented a tool to construct bottom-up UPPAAL models from scenarios written in the graphical environment of I-SCORE. Thus, scenarios can intuitively be designed and their critical properties can now be automatically verified using the mature and efficient algorithms of verification (symbolic model checking) provided by the tool UPPAAL. We also presented an approach for the low-latency and real-time performance of scenarios by synthesizing validated scenarios into FPGAS. To the best of our knowledge, none of the existing models of IS supports the automatic verification of scenarios or provides an implementation that guarantees their real-time execution.

Related work. In the last years, some researchers have proposed models for the specification and verification of IS. For instance, in [6] and [5] the authors propose a semantics based on process calculi. However, no practical techniques were proposed for the verification and real-time performance of scenarios as the framework presented here.

Our framework is influenced by the works in [16], [17] and [18]. In [16], the authors define an automata based semantics for the programming language ORC (https://orc.csres.utexas.edu). This language has been proposed as a model to orchestrate distributed services that are subject to constraints on their execution. The authors also provide a systematic construction of the TA model from ORC programs, allowing the verification of critical properties using UPPAAL. In [17], a novel model of ANTESCOFO (http://repmus.ircam.fr/antescofo) is proposed using Parametric Timed Automata [19]. Roughly, ANTESCOFO is an automatic accompaniment system consisting of both a listening machine which recognizes in real-time the events

described by a score, and a reactive system which coordinates and schedules in real-time the accompaniment actions. The proposed semantics allows the verification of qualitative and quantitative properties. Moreover, it permits the inference of constraints on the parametric delays of events for ensuring the composer's requirements. In [18], the authors present a TA and synchronous data-flow based framework for modelling and verification of multi-clock train control systems. Additionally, the authors propose an implementation on FPGAs of the validated models.

Future work. The use of UPPAAL for specifying scenario's properties may be cumbersome for composers. Then, our next step is to develop a front-end to intuitively verify such properties. Currently, we are implementing a tool to automatically generate SYSTEMVERILOG code from the model. Therefore, we plan to use the formal specification language SVA (SystemVerilog Assertions) [20] to validate some properties of the generated code.

Nowadays, composers have increasingly needed to manipulate streams in their multimedia scenarios. In [7], the authors present a Colored Petri Net (CPN) [21] model for specifying interactive scenarios extended with the ability to handle data audio streams. We intend to increase the expressive power of our model by integrating the CPN semantics for handling streams into our TA semantics. In this way, we will be able to specify and verify some properties of the multimedia processes (i.e., textures in IS).

In I-SCORE, external applications (e.g., MAX/MSP and PURE DATA) are used to execute the multimedia processes. Then, by following the ideas presented in [22], we plan to implement a Fast Ethernet module in order to provide a low-rate and reliable communication between the FPGA platform and the external applications which are running on standard operating systems. To conclude, we plan to synthesize DSP programs into FPGAs basing on the work done in [23], in which the authors compile into VHDL, DSP programs written in the functional programming language FAUST (http://faust.grame.fr).

ACKNOWLEDGMENT

We thank the anonymous reviewers for their detailed comments that helped us to improve this paper. Also, we would like to thank Jean-Michaël Celerier for his valuable remarks about the paper. This work has been supported by the ANR project OSSIA (ANR-12-CORD-0024) and SCRIME (Studio de Création et de Recherche en Informatique et Musique Électroacoustique).

REFERENCES

[1] A. Allombert, "Aspects Temporels d'un Système de Partitions Musicales Interactives pour la Composition et l'Exécution," Ph.D. Thesis, Université de Bordeaux, 2009.

[2] M. Desainte-Catherine, A. Allombert, and G. Assayag, "Towards a hybrid temporal paradigm for musical composition and performance: The case of musical interpretation," *Computer Music Journal*, vol. 37, no. 2, pp. 61–72, 2013.

[3] P. Sénac, P. de Saqui-Sannes, and R. Willrich, "Hierarchical time stream petri net: A model for hypermedia systems," in *Application and Theory of Petri Nets*, ser. LNCS, vol. 935. Springer, 1995, pp. 451–470.

[4] R. Marczak, M. Desainte-Catherine, and A. Allombert, "Real-time temporal control of musical processes," in *The Third International Conferences on Advances in Multimedia*, 2011, pp. 12–17.

[5] M. Toro, M. Desainte-Catherine, and C. Rueda, "Formal semantics for interactive music scores: a framework to design, specify properties and execute interactive scenarios," *Journal of Mathematics and Music*, vol. 8, no. 1, pp. 93–112, 2014.

[6] C. Olarte and C. Rueda, "A Declarative Language for Dynamic Multimedia Interaction Systems," in *Mathematics and Computation in Music*. Springer Berlin Heidelberg, 2009, vol. 38, pp. 218–227.

[7] J. Arias, M. Desainte-Catherine, and C. Rueda, "Modelling Data Processing for Interactive Scores Using Coloured Petri Nets," in *14th International Conference On Applications Of Concurrency To System Design*, 2014, pp. 186–195.

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

[9] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, Dec. 1997.

[10] R. Milner, *Communication and concurrency*. Prentice Hall, 1989.

[11] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, Jan. 2004, pp. 200–236.

[12] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, San Francisco, CA, USA, 2007.

[13] J. Rodriguez-Andina, M. Moure, and M. Valdes, "Features, Design Tools, and Application Domains of FPGAs," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 4, pp. 1810–1823, Aug. 2007.

[14] E. Monmasson, L. Idkhajine, M. N. Cirstea, I. Bahri, A. Tisan, and M. W. Naouar, "FPGAs in Industrial Control Applications," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, pp. 224–243, 2011.

[15] R. Dubey, *Introduction to Embedded System Design Using Field Programmable Gate Arrays*. London: Springer London, 2009.

[16] J. S. Dong, Y. Liu, J. Sun, and X. Zhang, "Towards Verification of Computation Orchestration," *Formal Aspects of Computing*, vol. 26, no. 4, pp. 729–759, 2014.

[17] J. Echeveste, A. Cont, J.-L. Giavitto, and F. Jacquemard, "Operational Semantics of a Domain Specific Language for Real Time Musician–Computer Interaction," *Discrete Event Dynamic Systems*, vol. 23, no. 4, pp. 343–383, Dec. 2013.

[18] Y. Jiang, H. Zhang, Z. Li, Y. Deng, X. Song, M. Gu, and J. Sun, "Design and Optimization of Multiclocked Embedded Systems Using Formal Techniques," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 2, pp. 1270–1278, 2015.

[19] R. Alur, T. A. Henzinger, and M. Y. Vardi, "Parametric real-time reasoning," in *Proceedings of the 25th annual ACM symposium on Theory of computing*. ACM Press, 1993, pp. 592–601.

[20] E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny, *The Power of Assertions in SystemVerilog*, 2010.

[21] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Berlin Heidelberg, 2009.

[22] R. Aviziensis, A. Freed, T. Suzuki, and D. Wessel, "Scalable Connectivity Processor for Computer Music Performance Systems," in *International Computer Music Conference*, 2000.

[23] R. Trausmuth, C. Dusek, and Y. Orlarey, "Using FAUST for FPGA Programming," in *9th International Conference on Digital Audio Effects*, 2006, pp. 18–20.