

Foundations for Reliable and Flexible Interactive Multimedia Scores

Jaime Arias, Myriam Desainte-Catherine, Carlos Olarte, Camilo Rueda

▶ **To cite this version:**

Jaime Arias, Myriam Desainte-Catherine, Carlos Olarte, Camilo Rueda. Foundations for Reliable and Flexible Interactive Multimedia Scores. Fifth Biennial International Conference on Mathematics and Computation in Music, Jun 2015, London, United Kingdom. pp.29-41, 10.1007/978-3-319-20603-5_3 . hal-01129394

HAL Id: hal-01129394
https://hal.science/hal-01129394

Submitted on 22 Sep 2015

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.

Foundations for Reliable and Flexible Interactive Multimedia Scores

Jaime Arias[1], Myriam Desainte-Catherine[1], Carlos Olarte[2], and Camilo Rueda[3]

[1] Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France
CNRS, LaBRI, UMR 5800, F-33400 Talence, France
[2] ECT, Universidade Federal do Rio Grande do Norte, Brazil.
[3] DECC, Pontificia Universidad Javeriana Cali, Colombia

Abstract. Interactive Scores (IS) is a formalism for composing and performing interactive multimedia scores with several applications in video games, live performance installations, and virtual museums. The composer defines the temporal organization of the score by asserting temporal relations (TRs) between temporal objects (TOs). At execution time, the performer may modify the start/stop times of the TOs by triggering interaction points and the system guarantees that all the TRs are satisfied. Implementations of IS and formal models of their behavior have already been proposed, but these do not provide usable means to reason about their properties. In this paper we introduce ReactiveIS, a programming language that fully captures the temporal structure of IS during both composition and execution. For that, we propose a semantics based on tree-like structures representing the execution state of the score at each point in time. The semantics captures the hierarchical aspects of IS and provides an intuitive representation of their execution. We also endow ReactiveIS with a logical semantics based on linear logic, thus widening the reasoning techniques available for IS. We show that ReactiveIS is general enough to capture the full behavior of IS and it also provides declarative ways to increase the expressivity of IS with, for instance, conditional statements and loops.

1 Introduction

Preliminaries. Interactive multimedia (e.g., live-performance arts) refers to computer-based design systems consisting of multimedia content that interacts with the performer's actions and other external events. Multimedia content is structured in a spatial and temporal order according to the author's requirements. The potential high complexity of these systems requires adequate specification languages for the complete description and verification of scenarios.

Interactive Scores (IS) [6] is a formalism for composing and performing interactive multimedia scores where the performer has the possibility to influence the execution of the score. This means the composer allows the performer to modify, during execution, the temporal organization of the score by adding *interaction*

points (IPs). 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 of the score. The IS model thus combines two temporal paradigms used in current multimedia tools [6]: *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 temporal relations between them. The time-flow paradigm is represented by the time at which the processes are actually executed.

Let us describe a simple example to introduce the terminology we shall use. In IS, boxes represent *temporal objects* (TOs) whose temporal organization is defined by asserting *temporal relations* (TRs) that those objects must obey. TRs define temporal (quantitative) and logical (qualitative) relations between TOs. More precisely, there are two qualitative relations that are defined between boxes: *precedence* and *posteriority*. Hence, TRs are enhanced with quantitative constraints by giving a range of possible durations in $[0, \infty]$. Consider for instance the IS on the left of Figure 1 which specifies the atmosphere of a cloud forest in a theatrical installation. The composer defines the score `S` in which the box `A` controls a machine that generates white smoke; the box `B` controls a group of fans that evenly distributes the smoke; all the boxes in the box `C` are performed once the smoke has been well distributed (defined by TRs r_3 and r_4); box `E` controls a set of lights in order to represent a sudden beam of light; finally, box `D` plays the sound of the howling of a wolf whose starting time depends on a performer's action (a mouse click).

TOs are classified into *textures* and *structures*. Textures represent the execution in time of a given multimedia process (e.g., changing the brightness of a light) while structures (i.e., the hierarchical organization of the score) represent the execution of a group of TOs with their own temporal organization. In our example, texture `A` has a duration of 2 time-units (`TU`) and it starts at `TU` 1 (relation r_1); structure `C` starts after 5 `TU` of stopping `A` (r_3) and after 3 `TU` of stopping `B` (r_4) and stops when textures `D` and `E` have finished (r_7, r_8); texture `D` starts when the message *"/mouse 1"* arrives between 2 and 5 `TU` after starting `C` (r_5); Finally, the score `S` finishes when `C` has finished (r_9).

In all executions of the score, the start time and duration of textures `A`, `B` and `E` do not change. Such TOs are seen as *static control points* that must be handled by the system without interaction with the environment. The start time of texture `D`, however, depends on triggering an IP. The starting of `D` modifies the duration of structures `C` and `S`. Hence, those TOs are controlled by *dynamic control points* that depend on the interaction with the environment. We also note that texture `D` starts automatically after 5 `TU` of starting `C` if the IP is not triggered. This *default action* guarantees that TRs are satisfied during execution.

The IS model is implemented in i-score (`http://i-score.org`), a tool that offers two different stages or times: *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 temporal properties. During the execution stage, the performer can dynamically trigger the IPs while the static

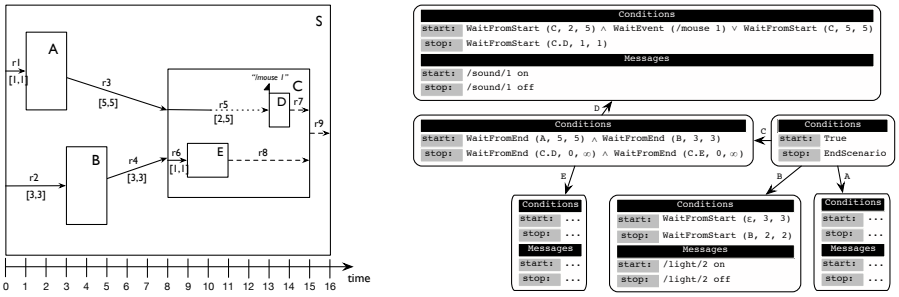


Fig. 1. Example of an interactive score and its program tree.

control points are triggered by the system. Since multimedia processes and IPs are handled by external applications, i-score uses multimedia protocols like OSC in order to send/receive the messages defined by the composer.

Scores in i-score are executed by the *ECO machine* [9] which is responsible of (1) triggering the static control points; (2) controlling the triggering of the dynamic control points; and (3) maintaining the temporal organization of the score. This machine relies on a Hierarchical Time Stream Petri Net (HTSPN) [12] to represent and execute the partially ordered set of events. Therefore, each time a score is written or modified, it must be translated into a HTSPN to be executed.

Motivation and Contributions. Some applications of IS such as video games, live performance installations, and virtual museum visits [1] demand two features that i-score as well as its execution model (HTSPN) do not currently support: (1) the use of more flexible control structures such as conditionals and loops [5]; and (2) scores must be verified before being played since they can be seen as critical systems where raise conditions (abnormal behaviors) should not happen. As an example of (1), consider a score where the composer may define an IP that decides between executing the TO `A` or `B`. As for (2), consider the situation where a given texture is never played due to inconsistent start/end conditions or a multimedia resource that receives an unexpected number of messages that it cannot handle concurrently. Dealing with (1) in i-score (i.e., on a horizontal time-line) would be hard and it would require a complete redesign of the HTSPN execution model. Moreover, due to the fact that there is one language to specify the score and another, completely different, to define the execution model, it does not seem trivial to define effective reasoning techniques to deal with (2).

In this paper we define ReactiveIS, a programming language that takes advantage and extends the full capacity of temporal organization during the composition and execution of IS. The syntax of ReactiveIS allows composers to define arbitrary hierarchies of processes and conditional commands –(1) above–. We endow the language with an operational semantics based on labelled trees that we claim to be simpler and more flexible than the current execution model in HTSPN. These structures allow to model the hierarchical aspect of IS and

provide an intuitive representation of their execution. Roughly, the program is represented by a tree whose nodes define the conditions needed to stop/start the TOs. The state of the system is a proper subtree of the program tree that contains information about the start/stop times of each TO. Hence, trees are considered as semantic and syntactic formal objects that are very close to the structure and behavior of IS. More interestingly, they can be defined and handled by means of a well-founded theory. This simple yet powerful characterization of IS allowed us to quickly develop an interpreter of ReactiveIS written in Ocaml. The tool produces a graphical representation of the execution of the IS as the one depicted in Figure 2.

In order to deal with (2) above, we give a declarative interpretation of ReactiveIS programs as formulas in intuitionistic linear logic (ILL) [7] with subexponentials [4]. We show that such interpretation is adequate: derivations in the logic correspond to traces of the program and vice-versa. Then, we can use all the meta-theory of ILL to reason about IS. In particular, we can verify whether an IS is free of raise conditions. Moreover, we can rely on the recent developments on the specification of temporal and spatial modalities in ILL (see [10]) to declaratively enrich ReactiveIS with new constructs. For instance, it would be possible to define IS whose hierarchy may change dynamically by allowing TOs to *move* into another TO according to the stimulus from the environment.

ReactiveIS thus offers the following advantages wrt to its predecessor i-score: 1) it offers an intuitive yet precise description of the behavior of IS; 2) the tree-based semantics gives a more concrete guidance to the implementer on how a score should be executed without dealing with the HTSPN model; 3) it is a first step towards a model for defining non-linear behavior (e.g., conditional statements) in IS; 4) the ILL characterization sets the basis for developing techniques and tools for the verification and analysis of IS.

Organization. Section 2 develops the theory of ReactiveIS: syntax, semantics and its properties (Sections 2.1, 2.2, 2.3). Section 2.4 is dedicated to the logical interpretation of programs and the kind of properties that can be verified. Section 2.5 discusses the ideas on how to extend the IS model to handle more flexible structures. Section 3 concludes the paper. Due to space restrictions, some auxiliary definitions and results appear in the extended version of this paper [3].

2 ReactiveIS: a Language for Specifying IS

In this section we introduce the syntax, semantics and logic characterization of ReactiveIS. We start with the constructors already available in i-score and later, in Section 2.5, we introduce the mechanisms for conditional statements.

Syntax. ReactiveIS programs are built from the following syntax:

$\langle score \rangle ::= \langle structure \rangle$
$\langle texture \rangle ::= \texttt{texture}(\langle params \rangle\ \langle msg \rangle\ \langle msg \rangle)$
$\langle structure \rangle ::= \texttt{structure}(\langle params \rangle\ \langle TO\text{-}list \rangle)$
$\langle params \rangle ::= \langle name \rangle\ \langle condition \rangle\ \langle condition \rangle$
$\langle TO\text{-}event \rangle ::= \texttt{start}\ \langle name \rangle \mid \texttt{end}\ \langle name \rangle$

$\langle condition \rangle ::= \texttt{wait}(\langle TO\text{-}event \rangle\ \langle min \rangle\ \langle max \rangle)$
$\quad \mid \texttt{event}\ \langle msg \rangle$
$\quad \mid (\langle condition \rangle \wedge \langle condition \rangle)$
$\quad \mid (\langle condition \rangle \vee \langle condition \rangle)$

Recall that a *structure* is a TO used to define the hierarchical organization of the score and a *texture* represents the execution of a given multimedia process by an external application. Hence, a *score* is a *structure* that represents the execution of a set of TOs (i.e., structures and textures). A structure is comprised of a set of parameters (explained below) and a (possibly empty) list of other TOs (TO-list). A texture requires, besides the parameters, two messages used to start and stop the external process. These messages are the output of the system and so they have to be sent to some other application by means of multimedia protocols such as OSC.

The syntactic unit *params* specifies a *name* (an identifier) for the TOs and also the starting and stopping *conditions*. Such conditions represent the TRs between TOs and define the temporal organization of the score.

Conditions in ReactiveIS can be: (1) `wait` conditions that define a delay from the start or from the end of a TO (*TO-event*). Delays are defined as a range between 0 and ∞, thus allowing flexibility in temporal specifications; (2) an `event` condition represents the triggering of a specific event by the environment. Such events are messages (*msg*), for instance *"/mouse 1"*, sent by the performer during execution (at IPs). Such messages represent the inputs of the system. More complex conditions can be written by using conjunctions and disjunctions.

As an example, consider the definition for the structure `C` in Figure 1:

```
1  Structure C = {
2   start.c = (Wait(End(A),5,5) & Wait(End(B),3,3));
3   stop.c  = (Wait(End(D),0,INF) & Wait(End(E),0,INF));
4   Texture D = {
5    start.c = ((Wait(Start(C),2,5) & Event("/mouse 1")) | Wait(Start(C),5,5));
6    stop.c  = Wait(Start(D),1,1);
7    start.msg  = "/sound/1 on"; stop.msg = "/sound/1 off";
8   }; ... };
```

Attributes `start.c` and `stop.c` represent, respectively, the start and stop conditions of the TO. The `Wait` condition receives three arguments: an event representing the start/end of a TO, its minimum and its maximum duration (that can be infinite, denoted `INF`). Condition `Event` receives a particular OSC message that will be sent by the performer (e.g., "*/mouse 1*"). If 5 time-units have elapsed after starting `C` and such message has not yet arrived, `D` will automatically start (due to the disjunction in the starting condition). Attributes `start.msg` and `stop.msg` specify the messages that must be sent to external multimedia processes.

2.1 Conditions and Program Representation

In this section we give a tree-based representation of ReactiveIS programs and we formalize the idea of *conditions*. Such definitions will be later used to describe the operational semantics of the language.

Conditions are built from a *Condition System* (CS) which is a first-order signature Σ that contains the distinguished predicates `WaitFromStart`, `WaitFromEnd`, `EndScenario` and `WaitEvent`. We also assume a (decidable) first-order theory Δ over Σ for dealing with deductions such as $x > 40 \models x > 0$. We shall use

$\mathcal{C}$ to denote the set of conditions (formulas) built from Σ and the grammar: $F, G, \ldots := \texttt{true} \mid A \mid F \wedge G \mid F \vee G$, i.e., conditions can be atomic formulas (e.g., predicates) or conjunctions/disjunctions of formulas.

A program in ReactiveIS is defined as a *labelled tree* whose nodes represent the TOs of the score. We will sometimes abuse notation and refer to TOs simply as nodes. Each node is associated with the conditions for starting and stopping the TO, and the corresponding messages.

Definition 1 (Program Tree). *Let $\mathcal{N}$ be a countable set of nodes, $\mathcal{B}$ the set of labels representing the names of* TO*s, and $\mathcal{M}$ the set of messages. A program tree is a labelled tree $P = \langle N, E, \ell, m, r\rangle$ where: $N \subseteq \mathcal{N}$ is the set of nodes; $E \subseteq N \times \mathcal{B} \times N$ is the set of edges; $\ell : N \to \mathcal{C} \times \mathcal{C}$ is a total function representing the start/end conditions; $m : N \rightharpoonup \mathcal{M} \times \mathcal{M}$ is a partial function representing the messages for starting/stopping an external application; and $r \in N$ is the root of the tree. Given $n \in N$, we shall use $c_s(n)$ and $c_e(n)$ (resp. $m_s(n)$ and $m_e(n)$) to to denote the starting/stopping conditions (resp. messages) for n.*

For a given tree T, the nodes, the edges, and the root node of T are denoted by $V(T)$, $E(T)$ and $root(T)$, respectively. We write $s \xrightarrow{a} t$ to represent an a-labeled edge from s (the source) to t (the target). As usual, sequences of labels $\alpha = a_0.a_1 \ldots a_n$ represent a path from the root r to a given node u in T. We use the empty sequence ε to represent the root of T. For a path p in T, $target_T(p)$ is its ending node.

The right part of Figure 1 shows a fragment of the program tree for our running example. The predicates `WaitFromStart(p,t1,t2)` and `WaitFromEnd(p,t1,t2)` hold when the time elapsed since the start and the end, respectively, of the target node of the path `p` is within the interval $[\texttt{t1}, \texttt{t2}]$. `WaitEvent(e)` waits for the external message `e`. Observe that the root node has no wait condition for starting (`true`) and it finishes when all its children have finished (`EndScenario`).

2.2 State Tree and Tree Operations

An execution state of a ReactiveIS program is also represented as a *labelled tree* that identifies the TOs currently being executed and the ones that have already stopped. Each node in the tree has associated the times on which the TO started and stopped. If a TO has not been stopped yet, we use as stop time the special symbol $\perp \notin \mathbb{Z}_+$. We shall use $\mathbb{Z}_\perp$ to denote $\mathbb{Z}_+ \cup \{\perp\}$.

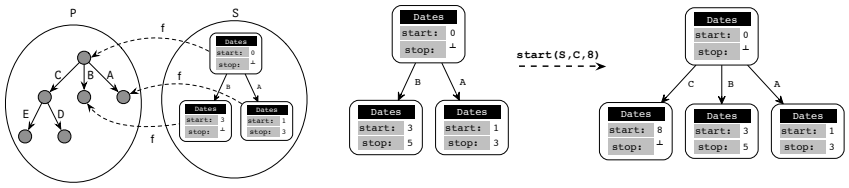
Definition 2 (State Tree). *A state tree is a labelled tree $S = \langle N, E, \ell, r\rangle$ where N, E and r are as in Definition 1 and $\ell : N \to \mathbb{Z}_+ \times \mathbb{Z}_\perp$ is a total function giving, for each node, its starting and ending times. Functions $t_s : N \to \mathbb{Z}_+$ and $t_e : N \to \mathbb{Z}_\perp$ give the starting and stopping time of a node, respectively.*

S is a *valid* state for a program tree P if S is homomorphic to P, i.e., there exists $f : V(S) \to V(P)$ that preservers the structure: $f(root(S)) = root(P)$ and $s \xrightarrow{a} t \in E(S)$ iff $f(s) \xrightarrow{a} f(t) \in E(P)$ (see Figure 2(a)).

Now we define two operations on state trees, stopping and starting a TO. We use $L(S)$ to denote the set of all paths in S including ε.

Stopping a TO. When a node n is stopped, its stop time, and the stop time of its (non already stopped) children, must be updated with the current time of execution. Formally, $stop(S, p, t) \triangleq \langle N, E, \ell \lhd \{n \mapsto (t_s(n), t) \mid n \in k\}, r\rangle$ where $k = \{\ n \mid n \in D(target_S(p)) \wedge t_e(n) = \bot\ \}$. "$\lhd$" means *relational overriding*, i.e, $\ell \lhd R \mathrel{\widehat{=}} R \cup \{\ x \mapsto y \mid x \mapsto y \in \ell \wedge x \notin dom(R)\}$; $D(v)$ denotes the set containing v and its descendants in S; $p \in L(S)$ is a path; and $t \in \mathbb{Z}_+$ is the current time of execution.

Starting a TO. This operation causes that a new b-labelled edge be added to the current state tree. This edge points to a new node having the current time as its start time and an undefined stop time. More precisely, for a non-empty path p, let $up(p)$ be the sequence of labels without the last label, and $last(p)$ be the last label of the sequence. For a path $p \in L(S)$ and a time $t \in \mathbb{Z}_+$, starting a TO is defined as $start(S, p, t) \triangleq \langle N \cup \{n_1\}, E \cup \{n \xrightarrow{b} n_1\}, \ell \cup \{n_1 \mapsto (t, \bot)\}, r\rangle$ where $n_1 \notin N, n = target_S(up(p)), b = last(p)$. Figure 2(b) shows the start of `C`.



(a) Program and State Tree (b) Starting a TO.

Fig. 2. Basic operations on state trees.

2.3 Operational Semantics

The semantics of ReactiveIS considers two kind of reduction relations, $\longrightarrow$ and $\Longrightarrow$, parametric on the program tree P (see Figure 3). Recall that the input of the program is a set of messages produced by the environment and the output is the set of messages the program must produce during a time-unit. Hence, the *observable* transition $\langle S, t\rangle \overset{I,O}{\Longrightarrow}_P \langle S', t+1\rangle$ means that at time t, the state tree S on input I reduces in one *time unit* to S' and outputs O. The observable transitions are obtained from finite sequences of internal transitions. Such *internal* transitions represent how the state S is gradually updated by starting/stopping TOs. It is important to notice that the changes in the state of the score are only visible at the end of the time-unit, i.e., it is assumed that internal transitions cannot be directly observed.

The internal transition $\langle St, O\rangle_S^{I,t} \longrightarrow_P \langle St', O'\rangle_S^{I,t}$ means that, given that the input in the current time-unit is I and the initial state is S, the state St moves to St' possibly adding new messages to the set O leading to O'. Let us give some

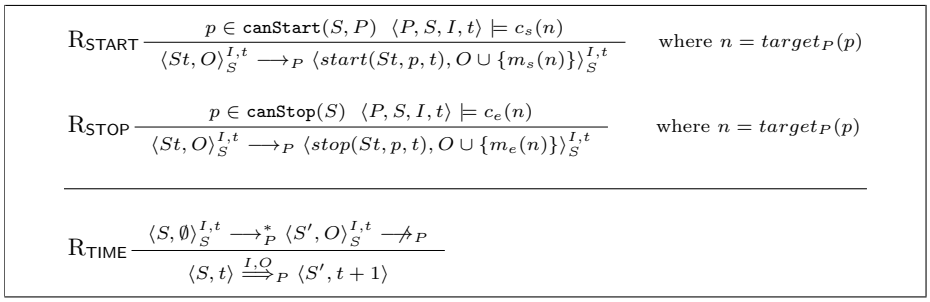


Fig. 3. Rules for the internal reduction $\longrightarrow$ and the observable reduction $\Longrightarrow$.

intuition. We define $p_{alive}(S) = \{p \,|\, p \in L(S) \wedge t_e(target_S(p)) = \bot\}$, i.e., the set of TOs that are currently running. Moreover, let $Children(p)$ be the set of paths of a program tree P from the root node to the children of the ending node of p (i.e., $target_P(p)$). Since a TO can only start if its parent is running and it has not stopped yet, we can compute the set of TOs that can start by defining $canStart(P,S) \triangleq \{p \,|\, p_{parent} \in p_{alive}(S) \wedge p \in Children(p_{parent})\} \setminus L(S)$.

The rule $\mathsf{R_{Start}}$ says that a TO is executed only if (a) it has not yet been executed and (b) its start condition is satisfied. Premise (a) is ensured with the aid of the set $canStart(S,P)$ explained before. Premise (b) is asserted by means of the relation $\langle P,S,I,t\rangle \models F$ that intuitively means that the current state satisfies the condition F. The precise definition of $\models$ is in [3].

The rule $\mathsf{R_{STOP}}$ dictates that a TO is stopped only if (a) it is currently being executed and (b) its end condition is satisfied. Premise (b) is similar as in the previous rule. Premise (a) is ensured with the aid of the set $canStop(S) \triangleq \{p \mid p \in L(S) \wedge t_e(target_S(p)) = \bot\}$ that contains the nodes in the state tree whose end time is not defined.

The only non-determinism of ReactiveIS programs is due to the signals provided by the environment. Then, we can prove that the observable relation is indeed a function (the proof is in [3]).

Theorem 1 (Determinism). *For all state S and input I, if $\langle S,t\rangle \overset{I,O_1}{\Longrightarrow}_P \langle S_1', t'\rangle$ and $\langle S,t\rangle \overset{I,O_2}{\Longrightarrow}_P \langle S_2', t'\rangle$ then $O_1 = O_2$ and $S_1' = S_2'$.*

2.4 Logical Characterization of IS

An appealing feature of ReactiveIS is that it allows a logic characterization as formulas in intuitionistic linear logic (ILL) [7] with subexponentials [4] (SELL). The technical details of this semantics can be found at [3]. In the following, we shall give some intuitions to understand what kind of properties we are able to verify about IS.

The formula $!^a F$ in SELL means that F is marked with a given modality a. The index a is taken from a poset $\langle I, \preceq\rangle$ (the subexponential signature) and it can be interpreted as a spatial location or a time-unit [10]. Here, we shall mark

the formulas with subexponentials of the form $t.x$ where t represents the current time-unit and "x" can be:

- $!^{t.i}F$: F is an input from the environment, e.g., $!^{t.i}\texttt{evt}(\texttt{mouse1})$
- $!^{t.o}F$: F is an observable action, e.g., $!^{t.o}\texttt{msg}(m)$ means that the starting/stopping message m was added.
- $!^{t.s.p}F$: F represents information about the state, e.g., $!^{t.s.A}\texttt{box}(-,-)$ means that `A` has not been started yet. We use "$-$" instead of "$\bot$", as in the previous section, since "$\bot$" is a logical symbol in ILL.

The advantage of using subexponentials is that we can neatly split the logical context in a sequent. In our particular case, the context is split into different time-units and each time-unit stores information about inputs from the environment ($t.i$), observable actions ($t.o$) and information about the state of the system ($t.s$). To better understand this idea, consider the following derivation:

$$\frac{!^{4.i}\texttt{evt}(\text{e2}), !^{4.i}\texttt{evt}(\text{e3}) \longrightarrow !^{4.i}\texttt{evt}(\text{e3})}{!^{3.i}\texttt{evt}(\text{e1}), !^{4.i}\texttt{evt}(\text{e2}), !^{4.i}\texttt{evt}(\text{e3}), !^{4.s.A}\texttt{box}(5{,}7) \longrightarrow !^{4.i}\texttt{evt}(\text{e3})}\ !_R$$

Roughly, we are trying to prove that the event $e3$ occurred in the time-unit 4. The introduction rule for ! ($!_R$, called promotion rule) forces to delete (weaken) from the context all the formulas with subexponentials not related to $4.i$. Then, we cannot use the information available on time-unit 3 (i.e., $!^{3.i}\texttt{evt}(\text{e1})$) nor the information about the state of the system (i.e., $!^{4.s.A}\texttt{box}(5,7)$).

The encoding of each TO in a ReactiveIS program gives rise to three formulas. Namely, `ctr` to control when to start/stop the TO and `str` and `stp` to handle the action of starting/stopping the TO. Let us consider the texture `D` in our running example whose starting condition depends on the starting of `C` and an event from the environment. We define the control formula as follows:

$$\begin{aligned}\texttt{ctr(D,t)} \stackrel{\text{def}}{=}\ & !^{t.s.D}\texttt{P_STOP} \multimap \texttt{stop-imm}(D,t)\ \&\ !^{t.s.D}\texttt{P_RUN} \multimap \texttt{decide}(D,t)\ \&\\ & !^{t.s.D}\texttt{P_IDLE} \multimap \forall n,m.(!^{t.s.D}\texttt{box}(n,m) \multimap !^{(t+1).s.D}\texttt{box}(n,m))\end{aligned}$$

Intuitively, `D` can only proceed if its parent `C` has already added to the context one of the predicates `P_STOP`, `P_RUN` or `P_IDLE` notifying its current state (stopped, currently running or idle –already stopped or not started–). We recall that $F \multimap G$ represents linear implication. The additive conjunction & allows us to choose between three possible choices: stop immediately (`stop-imm`), decide to start or stop (`decide`) or continue in the same state. The definition of these formulas can be found in [3].

Let us consider the formulas needed to start the execution of `D`:

$$\begin{aligned}&\texttt{str(D,t)} \stackrel{\text{def}}{=} (\texttt{condition} \multimap \texttt{start}(D,t))\ \&\ (\texttt{default} \multimap !^{(t+1).s.D}\texttt{box}(-,-))\\ &\texttt{start}(D,t) \stackrel{\text{def}}{=} !^{t.s.D}\texttt{box}(t,-) \otimes !^{(t+1).s.D}\texttt{box}(t,-) \otimes !^{t.o}\texttt{msg}(m_s(D))\end{aligned}$$

The formula `condition` is obtained by translating the starting condition of `D` into a SELL formula. The formula `default` (see [3]) says that the starting conditions cannot be satisfied in the current time-unit and then, the state of `D` remains the same. Note that the formula $\texttt{start}(D,t)$ adds to the context the

information needed to deduce that D started at time-unit t and it also adds to the context $t.o$ the starting message of `D`. The formulas controlling the stopping of `D` can be defined similarly.

With the aid of these formulas and some auxiliary definitions in [3], we can define an encoding $[\![\cdot]\!]$ mapping ReactiveIS programs into SELL formulas. Moreover, by relying on a focused [2] proof system for SELL, we can show that operational steps correspond to derivations in SELL and vice-versa (see theorem below). Focusing is a discipline on proofs to reduce the non-determinism during proof search. Hence, focused proofs can be interpreted as the normal form proofs for proof search. Roughly, once we choose to work on a formula (i.e., we focus on it), we do not have more choices that decompose it completely. Hence, in the end of the focused phase, we observe that the logical derivation mimicked exactly the operational steps of the encoded program.

Theorem 2 (Adequacy). *Let P be a* ReactiveIS *program. Then, $\langle S, t\rangle \overset{I,O}{\Longrightarrow}_P \langle S', t+1\rangle$ iff the sequent $[\![P]\!], [\![S]\!]_t, [\![I]\!]_t \longrightarrow [\![S']\!]_{t+1} \otimes [\![O]\!]_t$ is provable in* SELL.

The previous theorem opens the possibility of reasoning about IS by using well established techniques in proof theory. Moreover, all the tools developed for SELL [10] can be applied to verify properties of scores. For that, we consider SELL sequents of the form $[\![P]\!], [\![S_{\mathtt{init}}]\!]_0, \mathtt{env} \longrightarrow \mathcal{G}$ where P is the program (the score); $S_{\mathtt{init}} = \langle \{r\}, \emptyset, \{r \mapsto (0, \bot)\}, r\rangle$ is the initial configuration of the score; `env` encodes any possible input from the environment (i.e., a disjunction of all the possible inputs from the environment); and $\mathcal{G}$ encodes the property to be verified (i.e., the goal).

Let us give some examples of $\mathcal{G}$-formulas. Consider the case where we want to verify that two TOs `A` and `B` must be executed concurrently. Then we can set $\mathcal{G} = \Cup t.\exists n, m.!^{t.s.A}\mathtt{box}(n, -) \otimes !^{t.s.B}\mathtt{box}(m, -)$ meaning that there exists a time-unit t ($\Cup t$) such that in that time-unit `A` and `B` have already started and not stopped. As another example, consider the fact that regardless the inputs from the environment, the TO `B` cannot be currently playing if `A` has finished its execution. In that case, we have $\mathcal{G} = \Cap t.\exists n_1, n_2.!^{t.s.A}\mathtt{box}(n_1, n_2) \multimap \exists n_1', n_2'.!^{t.s.B}\mathtt{box}(n_1', n_2')$ meaning that for all time-unit t ($\Cap t$), if `A` has already stopped, then it must be the case that `B` has already stopped too. Now assume that there is a precedence relation between `A` and `B`, i.e., `B` cannot start if `A` is currently playing. In this case, $\mathcal{G} = \Cap t.(\exists n.!^{t.s.A}\mathtt{box}(n, -)) \multimap !^{t.s.B}\mathtt{box}(-, -)$. Finally, one may be interested in proving that there exists at least one execution path such that a given TO A is executed. This can be formalized by proving the property $\mathcal{G} = \Cup t.\exists n, m.!^{t.s.A}\mathtt{box}(n, m)$.

2.5 Extending the IS Model

Having a formal model for IS opens the possibility to propose new programming constructs and reason about its behavior. For instance, it turns out that the notion of conditions as logical formulas and its realization in the operational semantics ($\langle P, S, I, t\rangle \models F$) are general enough to define conditional statements

in ReactiveIS. For that, let us extend the syntax of ReactiveIS to consider conditions of the form $\langle conditions \rangle ::= \ldots \mid event\ (e(n)\ op\ \langle value \rangle)$ where e is an event with a carried value n and op is a relational operator (e.g., $\leq$, $=$, etc). Let *mouse* be a parametric event with two possible values, 1 and 2. By defining the starting condition of a TO `A` as `event`(`mouse`$(n) = 1$), and the condition of `B` as `event`(`mouse`$(n) = 2$), we can define a score that waits until the *mouse* event is detected. Then, it decides whether to execute `A` or `B`. Since the conditions on the carried value n are mutually exclusive, it would be more natural to write `if` c `then` `A` `else` `B` to specify such behavior. Hence, IPs can now be used to express non-linear behaviors: once an event is detected, the evaluation of the expression "*n op value*" will determine the flow of the score.

Interestingly, the logical semantics of ReactiveIS will allow the composer to verify that, regardless the path taken by the performer, the desired properties of the score hold. These techniques can also be used to avoid mistakes during composition. For instance, consider the situation where the starting condition of another TO `C` depends on both the starting of `A` and `B`. In this case, the logical semantics will detect that `C` will be never played.

Loops can be also obtained in a similar fashion. However, special attention must be paid to avoid two copies of the same TO during execution. For that it suffices to consider that the repetition of a TO `A` is restricted to: (1) `A` has already stopped and (2) the performer must send an event to control whether `A` has to be played again (i.e., loops are guarded by IPs). The semantics and the logic characterization require a minimal change to reset the starting and stopping times of `A` in order to enable its next execution.

3 Concluding Remarks and Related Work

We have introduced ReactiveIS, a new programming language for the composition and performing of interactive scores. We defined an operational semantics for ReactiveIS based on labelled trees which is simpler and more intuitive that the HTSPN model of i-score. We also endowed ReactiveIS with a declarative semantics based on intuitionistic linear logic with subexponentials, thus allowing us to prove the correct behavior of scores. ReactiveIS then strives at setting the foundations for reasoning about IS and, more importantly, to extend its functionality in a declarative way.

The idea of a tree-based semantics is influenced by the works in [8] and [14]. In [8] a semantics for Orc, a language to specify programs to orchestrate the invocation of sites that are subject to constraints on their execution, is presented. Such semantics considers trees annotated with information about the values of the program variables and the times at which they are assigned. In [14] a graph model to represent the semantics of video (VideoGraph) is presented. In such model the nodes represent events and they are linked to each other based on their containment and temporal relationships. On the other side, our logical characterization of ReactiveIS is based on the ideas in [10] where subexponentials in linear logic were used to give logical semantics to concurrent programming

languages featuring modalities. In [11,13] the authors propose a semantics for IS based on process calculi. However, no practical techniques were proposed for the verification of the score as the logical characterization presented here. Moreover, the models proposed in [11, 13] cannot be straightforwardly extended to deal with non-linear behavior.

Our next step is to formalize, for instance in Coq, the semantics presented here in order to generate a verified interpreter. Since the use of SELL formulas for specifying score's properties may be cumbersome for non-experts, we plan to develop a front-end, e.g., a small assertion language, to express such properties.

References

1. Allombert, A., Marczak, R., Desainte-Catherine, M., Baltazar, P., GarnierLaurent: Virage: Designing An Interactive Intermedia Sequencer From Users Requirements And Theoretical Background. In: International Computer Music Conference (2010)
2. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. J. Log. Comput. 2(3), 297–347 (1992)
3. Arias, J., Desainte-Catherine, M., Olarte, C., Rueda, C.: Foundations for reliable and flexible interactive multimedia scores. Tech. Rep., LaBRI, University of Bordeaux (Mar 2015)
4. Danos, V., Joinet, J., Schellinx, H.: The structure of exponentials: Uncovering the dynamics of linear logic proofs. In Proc. of Computational Logic and Proof Theory, Third Kurt Gödel Colloquium, KGC'93. Lecture Notes in Computer Science, vol. 713, pp. 159–171. Springer (1993)
5. De la Hogue, T., Baltazar, P., Desainte-Catherine, M., Chao, J., Bossut, C.: OSSIA: Open Scenario System for Interactive Applications. In: Journées d'Informatique Musicale. pp. 78–84. Bourges (2014)
6. Desainte-Catherine, M., Allombert, A., Assayag, G.: Towards a hybrid temporal paradigm for musical composition and performance: The case of musical interpretation. Computer Music Journal 37(2), 61–72 (2013)
7. Girard, J.: Linear logic. Theor. Comput. Sci. 50, 1–102 (1987)
8. Hoare, T., Menzel, G., Misra, J.: A tree semantics of an orchestration language. In: Engineering Theories of Software Intensive Systems. NATO Science Series II: Mathematics, Physics and Chemistry, vol. 195, pp. 331–350. Springer (2005)
9. Marczak, R., Desainte-Catherine, M., Allombert, A.: Real-time temporal control of musical processes. In. Proc of the Third International Conferences on Advances in Multimedia. pp. 12–17. MMEDIA 2011 (2011)
10. Nigam, V., Olarte, C., Pimentel, E.: A general proof system for modalities in concurrent constraint programming. In Proc. of CONCUR. Lecture Notes in Computer Science, vol. 8052, pp. 410–424. Springer (2013)
11. Olarte, C., Rueda, C.: A Declarative Language for Dynamic Multimedia Interaction Systems. In Proc. of Mathematics and Computation in Music. Communications in Computer and Information Science, vol. 38, pp. 218–227. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
12. Sénac, P., de Saqui-Sannes, P., Willrich, R.: Hierarchical time stream petri net: A model for hypermedia systems. In Proc. of Application and Theory of Petri Nets. Lecture Notes in Computer Science, vol. 935, pp. 451–470. Springer (1995)

13. Toro, M., Desainte-Catherine, M., Rueda, C.: Formal semantics for interactive music scores: a framework to design, specify properties and execute interactive scenarios. Journal of Mathematics and Music 8(1), 93–112 (2014)
14. Tran, D.A., Hua, K.A., Vu, K.: Videograph: A graphical object-based model for representing and querying video data. In: ER. pp. 383–396 (2000)

Appendix

We present here some technical details about the operational semantics of ReactiveIS (Section A) and its logical characterization (Section B).

A Operational Semantics

In this section we introduce some definitions and auxiliary results needed to define the operational semantics of ReactiveIS.

The following definition formalizes the idea of when a *condition* can be satisfied by the current information. We consider configurations of the form $\langle P, S, I, t\rangle$ where P is the program, S the state of the score, I the input (set of messages) and t the current time unit. The assertion $\langle P, S, I, t\rangle \models F$ means that the configuration satisfies the condition F (recall that F are built from a Condition System (see Section 2.1)).

Definition 3 (Semantics of $\models$).

$\langle P, S, I, t\rangle \models \texttt{true}$

$\langle P, S, I, t\rangle \models \texttt{WaitFromStart}(p, t_1, t_2)$ *iff*
$$\exists n \cdot n \in V(S) \wedge n = target_S(p) \wedge t_1 \leq t - t_s(n) \leq t_2$$

$\langle P, S, I, t\rangle \models \texttt{WaitFromEnd}(p, t_1, t_2)$ *iff*
$$\exists n \cdot n \in V(S) \wedge n = target_S(p) \wedge t_e(n) \neq \bot \wedge t_1 \leq t - t_e(n) \leq t_2$$

$\langle P, S, I, t\rangle \models \texttt{EndScenario}$ *iff*
$$\forall p \cdot p \in out_P(root(P)) \Rightarrow t_e(target_S(p)) \neq \bot$$

$\langle P, S, I, t\rangle \models \texttt{WaitEvent(e)}$ *iff* $e \in I$

$\langle P, S, I, t\rangle \models F \wedge G$ *iff* $\langle P, S, I, t\rangle \models F$ *and* $\langle P, S, I, t\rangle \models G$

$\langle P, S, I, t\rangle \models F \vee G$ *iff* $\langle P, S, I, t\rangle \models F$ *or* $\langle P, S, I, t\rangle \models G$

In the following, we state some basic properties of the internal relation $\longrightarrow$ on the operational semantics.

Proposition 1 (Monotonicity). *For any* ReactiveIS *program P and valid state S, if $\langle St, O\rangle_S^{I,t} \longrightarrow_P \langle St', O'\rangle_S^{I,t}$ then:*

1. *$O \subseteq O'$*
2. *St is homomorphic to St'. Moreover, St' is a valid state of P (i.e., St' is homomorphic to P).*

Proof. The proof proceeds by induction on the derivation $\longrightarrow_P$ with case analysis on the last rule applied. By simple inspection, we know that rules $\mathrm{R}_{\mathsf{START}}$ and $\mathrm{R}_{\mathsf{STOP}}$ only add elements to O. Then (1) holds. As for (2), if S is a valid state of P, then there exists a homomorphism f relating S and P. Let us analyze the rule $\mathrm{R}_{\mathsf{START}}$. Assume that p is the path of a TO to be started. By definition of *canStart*, we know that the parent of the ending node of p is currently being executed. Moreover, by definition of $start(S, p, t)$, the node denoted by p is located right below its parent (since p is a path in the tree). Hence, there exists a homomorphism f' between St' and St. In rule $\mathrm{R}_{\mathsf{STOP}}$, the set of nodes and edges is not modified (only the stop information of the ending node of p). Then, trivially St is homomorphic to St'. □

In the following, we shall use γ, γ' to range over configurations of the form $\langle St, O\rangle_S^{I,t}$. We shall say that a TO p is *enabled* in a configuration γ if p triggers a STOP/START reduction. The next observation shows that firing an event during a time-unit does not disable other events. This fact will be later used to prove that ReactiveIS is deterministic.

Observation 1 (TO-Potentiality)**.** *Consider a configuration γ where two different* TO*s p, p' are enabled. Assume also that $\gamma \longrightarrow_P \gamma'$ using the* TO *p. Then:*

1. *p is not enabled at γ', and*
2. *p' is enabled at γ iff p' is enabled at γ'.*

Proof. To prove (1), note that operations $start(S, p, t)$ and $stop(S, p, t)$ (see Section 2.2) guarantee that p cannot be started/stopped again (see *canStop* and *canStart* predicates). As for (2), note that the enabled conditions depend only on the initial state S. Hence, p' is enabled at γ iff it is enabled at γ'. □

Lemma 1 (Confluence). *For any* ReactiveIS *program P and valid state St, if $\langle St, O\rangle_S^{I,t} \longrightarrow_P \gamma_1$, $\langle St, O\rangle_S^{I,t} \longrightarrow_P \gamma_2$ and $\gamma_1 \neq \gamma_2$ then there exists γ_3 such that $\gamma_1 \longrightarrow_P \gamma_3$ and $\gamma_2 \longrightarrow_P \gamma_3$.*

Proof. Assume that $\langle St, O\rangle_S^{I,t} \longrightarrow_P \gamma_1$, $\langle St, O\rangle_S^{I,t} \longrightarrow_P \gamma_2$. If $\gamma_1 \neq \gamma_2$ we have to consider 4 cases: both reductions are STOP-reductions; both reductions are START-reductions; one reduction corresponds to the START rule the other to the STOP rule and vice versa. In the first two cases, since $\gamma_1 \neq \gamma_2$, it must be the case that the selected TO p in the reductions is different. By using Observation 1, we can show that there exists γ_3 such that $\gamma_1 \longrightarrow_P \gamma_3$ and $\gamma_2 \longrightarrow_P \gamma_3$. □

Corollary 1 (Determinism). *For all state S and input I, if $\langle S, t\rangle \stackrel{I,O_1}{\Longrightarrow}_P \langle S_1', t'\rangle$ and $\langle S, t\rangle \stackrel{I,O_2}{\Longrightarrow}_P \langle S_2', t'\rangle$ then $O_1 = O_2$ and $S_1' = S_2'$.*

Proof. Directly from Lemma 1. □

B Logical Characterization of ReactiveIS

In this section we present at length the logical characterization of ReactiveIS programs as formulas in SELL [10]. We shall use the notations and definitions in Section 2, e.g., function $m_s(\cdot), m_e(\cdot), c_s(\cdot), c_e(\cdot)$, etc.

Subexponential Signature. First we need to define the subexponential signature $\langle I, \preceq \rangle$. As we explained in Section 2.4, we shall use subexponential indexes of the form $t.i$, $t.o$, $t.s$ to mark formulas related to the environment (the inputs), the observable actions (the outputs) and the state of the system respectively. Following [10], the structure of the subexponentials to deal with temporal modalities must consider subexponentials of the shape i and $i+$. The former represents a given-time unit i. The latter is used to store formulas valid from the time-unit i on. The structure is depicted in Figure 4. Note that the subexponentials of the shape $t.i$ and $t.o$ are unrelated. The subexponentials of the shape $t.s$ preserve the hierarchical structure of the score. For instance, in our running example, the TO `C` have two children `D` and `E`. As we shall see in brief, this will allow `C` to control the behavior of `D` and `E`. The subexponential TI (which is greater than any $t.i$) will be used to define the encoding of the environment. Finally, the subexponential TP (which is grater than any $t.p$) will be used to store the encoding of the TOs.

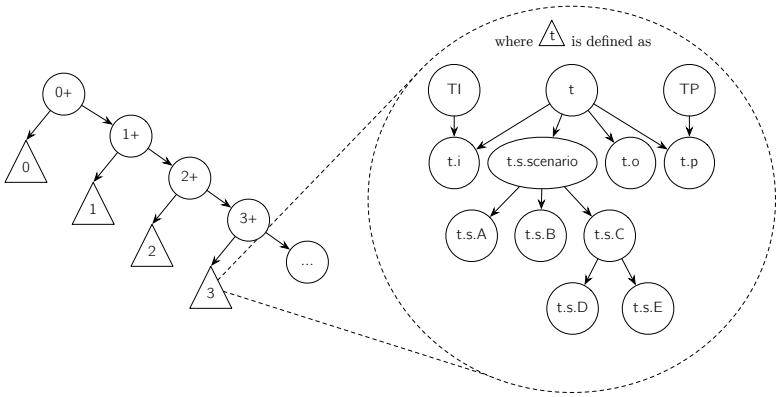


Fig. 4. Subexponential structure $\langle I, \preceq \rangle$. $a \to b$ means $b \preceq a$.

Input-Output. Let us start encoding the inputs and outputs of ReactiveIS, i.e., the set of messages the program can input and output. For any message m_i we define a constant symbol `m_i` (e.g., `mouse1`). We also consider the unary predicates `evt`$(\cdot)$ and `msg`$(\cdot)$ to represent, respectively, the fact that an input and an output have been added. Hence, a set of input (resp. output) messages

$I = \{m_1, m_2, \cdots, m_n\}$ (resp. $O = \{m'_1, m'_2, \cdots, m'_m\}$) is encoded in SELL as:

$$\begin{aligned}[\![I]\!]_t &= !^{t.i}\texttt{evt}(msg_1) \otimes !^{t.i}\texttt{evt}(msg_2) \otimes \cdots !^{t.i}\texttt{evt}(msg_n)\\ [\![O]\!]_t &= !^{t.o}\texttt{msg}(m'_1) \otimes !^{t.o}\texttt{msg}(m'_2) \otimes \cdots !^{t.o}\texttt{msg}(m'_m)\end{aligned}$$

Intuitively, the messages from the set I (resp. O) are available in the logical context $t.i$ (resp. $t.o$).

Encoding Textures. The encoding of a TO defines three kind of formulas: `ctr` to control when to start/stop the TO and `str` and `stp` to handle the action of starting/stopping the TO. Such formulas modify the state of the TO in the current time-unit and define the state of the TO for the next time-unit. The interpretation of textures and structures is similar. However, in the case of a structure `S`, we need to control also the execution of `S`'s children.

We start defining the aforementioned formulas for a given texture `A`:

$$\begin{aligned}\texttt{ctr}(A,t) \stackrel{\text{def}}{=}\ & !^{t.s.A}\texttt{P_STOP} \multimap \texttt{stop-imm}(A,t)\&\\ & !^{t.s.A}\texttt{P_RUN} \multimap \texttt{decide}(A,t)\&\\ & !^{t.s.A}\texttt{P_IDLE} \multimap \forall n,m.(!^{t.s.A}\texttt{box}(n,m) \multimap !^{(t+1).s.A}\texttt{box}(n,m))\end{aligned}$$

where:

$$\begin{aligned}\texttt{stop-imm}(A,t) \stackrel{\text{def}}{=}\ & \forall n,m.(!^{t.s.A}\texttt{box}(n,m) \multimap\\ & \quad n = - \multimap (!^{t.s.A}\texttt{box}(-,-) \otimes !^{(t+1).s.A}\texttt{box}(-,-))\&\\ & \quad n \neq - \multimap (!^{t.s.A}\texttt{box}(n,t) \otimes !^{(t+1).s.A}\texttt{box}(n,t) \otimes !^{t.o}\texttt{msg}(m_e(A))))\\ \texttt{decide}(A,t) \stackrel{\text{def}}{=}\ & (!^{t.s.A}\texttt{box}(-,-) \multimap \texttt{str}(A,t))\&\\ & \forall n.(!^{t.s.A}\texttt{box}(n,-) \multimap \texttt{stp}(A,t)\&\\ & \forall n,m.(!^{t.s.A}\texttt{box}(n,m) \multimap\\ & \quad !^{t.s.A}\texttt{box}(n,m) \otimes !^{(t+1).s.A}\texttt{box}(n,m))\end{aligned}$$

The predicate `P_STOP` is added by the parent of `A` to signal that `A` must stop immediately. As we shall see, this happens when the parent of `A` stops at time-unit t. `P_RUN` says that the parent of `A` is currently running and `P_IDLE` signals that the parent of `A` has already stopped or it has not started yet. Hence, the formula `ctr` verifies first what was the decision of `A`'s parent and proceeds accordingly: it stops immediately, it decides whether to start or to stop or it simply copies the state to the next time-unit.

The formula `stop-imm` simple updates the stopping time of `A`: if `A` has not already started, then the starting/stopping time will be "-". Otherwise, the starting time-remains the same and the stopping time is updated to t. Moreover, the stopping message of `A` ($m_e(A)$) is added to the $t.o$ context ($!^{t.o}\texttt{msg}(m_e(A))$).

The formula `decide` checks whether `A` has not already started yet ($\texttt{box}(-,-)$). In that case, `A` can start (`str` defined below). If in the current time-unit one can deduce $\texttt{box}(n,-)$ for some n, then `A` may stop (`stp` defined below). Finally, if the stop and start time are already defined for A, the formula `decide` simply copies that information to the next time-unit.

The formula controlling the start of A is:

$$\begin{aligned}
\texttt{str}(A,t) &\stackrel{\texttt{def}}{=} \texttt{condition} \multimap \texttt{start}(A,t)\& \\
&\quad\ \texttt{default} \multimap !^{(t+1).s.A}\texttt{box}(-,-) \\
\texttt{start}(A,t) &\stackrel{\texttt{def}}{=} !^{t.s.A}\texttt{box}(t,-) \otimes !^{(t+1).s.A}\texttt{box}(t,-) \otimes !^{t.o}\texttt{msg}(m_s(A))
\end{aligned}$$

The formula condition corresponds to the interpretation in SELL of the starting condition of A, $[\![c_s(A)]\!]_t$, where [4]:

$$\begin{aligned}
[\![\texttt{true}]\!]_t &= 1 \\
[\![F \wedge G]\!]_t &= [\![F]\!]_t \otimes [\![G]\!]_t \\
[\![F \vee G]\!]_t &= [\![F]\!]_t \oplus [\![G]\!]_t \\
[\![\texttt{WaitFromStart}(A,k,l)]\!]_t &= \exists n,m.(!^{t.s.A}\texttt{box}(n,m) \otimes n+k \leq t \otimes n+l \geq t) \\
[\![\texttt{WaitEvent}(e)]\!]_t &= !^{t.i}\texttt{evt}(\texttt{e})
\end{aligned}$$

Definitions for the predicates WaitFromEnd, and EndScenario are similar. The formula default corresponds to the condition when none of the starting conditions can be satisfied. Such formula corresponds to $[\![c_s(A)]\!]_t^{\perp}$ where:

$$\begin{aligned}
[\![\texttt{true}]\!]_t^{\perp} &= 0 \\
[\![F \wedge G]\!]_t &= [\![F]\!]_t^{\perp} \oplus [\![G]\!]_t^{\perp} \\
[\![F \vee G]\!]_t &= [\![F]\!]_t^{\perp} \otimes [\![G]\!]_t^{\perp} \\
[\![\texttt{WaitFromStart}(A,k,l)]\!]_t &= !^{t.s.A}\texttt{box}(-,-) \oplus \\
&\quad\ \exists n,m.(!^{t.s.A}\texttt{box}(n,m) \otimes (n+k > t \oplus n+l < t)) \\
[\![\texttt{WaitEvent}(e)]\!]_t &= !^{t.i}\texttt{evt}^{\perp}(\texttt{e})
\end{aligned}$$

We note that the above definition of default requires that the environment (defined below) provides either that an event happened (e.g., $!^{t.s}\texttt{evt}(\texttt{mouse})$) or it did not happen (i.e., $!^{t.s}\texttt{evt}^{\perp}(\texttt{mouse})$).

The formulas defining how the TO have to be stopped are defined similarly:

$$\begin{aligned}
\texttt{stp}(A,t) &\stackrel{\texttt{def}}{=} \texttt{condition} \multimap \texttt{stop}(A,t)\& \\
&\quad\ \texttt{default} \multimap !^{(t+1).s.A}\texttt{box}(-,-) \\
\texttt{stop}(A,t) &\stackrel{\texttt{def}}{=} \forall n,m.!^{t.s.A}\texttt{box}(n,m) \multimap \\
&\qquad !^{t.s.A}\texttt{box}(n,t) \otimes !^{(t+1).s.A}\texttt{box}(n,t) \otimes !^{t.o}\texttt{msg}(m_e(A))
\end{aligned}$$

Encoding Structures. The encoding of a structure is similar to that of textures but it requires to take control of the execution of its children. For that, we modify the above definitions of $\texttt{start}(\cdot)$ and $\texttt{stop}(\cdot)$ as follows:

$$\begin{aligned}
\texttt{start}(A,t) &\stackrel{\texttt{def}}{=} !^{t.s.A}\texttt{box}(t,-) \otimes !^{(t+1).s.A}\texttt{box}(t,-) \otimes !^{t.o}\texttt{msg} \bigotimes\nolimits_{p \in suc(A)} !^{t.s.p}\texttt{P_RUN} \\
\texttt{stop}(A,t) &\stackrel{\texttt{def}}{=} \forall n,m.!^{t.s.A}\texttt{box}(n,m) \multimap !^{t.s.A}\texttt{box}(n,t) \otimes !^{(t+1).s.A}\texttt{box}(n,t) \\
&\quad \otimes \bigotimes\nolimits_{p \in suc(A)} !^{t.s.p}\texttt{P_STOP}
\end{aligned}$$

[4] We abuse notation by using the same symbol $[\![\cdot]\!]_t$ for different encodings: inputs, outputs, conditions, etc. However, from the context, it is easy to know the meaning of such function.

Observe that we add the predicates P_RUN and P_STOP to all the successors of A. Moreover, it can be the case that A cannot start in the current time-unit because its starting conditions do not hold. In this case, the successors of A must be notified that A is in an idle state:

$$\begin{aligned}\texttt{str}(A,t) \stackrel{\texttt{def}}{=}\ & \texttt{condition} \multimap \texttt{start}(A,t)\& \\ & \texttt{default} \multimap !^{(t+1).s.A}\texttt{box}(-,-) \bigotimes_{p\in suc(A)} !^{t.s.p}\texttt{P_IDLE}\end{aligned}$$

Similarly, if A cannot stop in the current time-unit, the successors must be notified that A is currently running:

$$\begin{aligned}\texttt{stp}(A,t) \stackrel{\texttt{def}}{=}\ & \texttt{condition} \multimap \texttt{stop}(A,t)\& \\ & \texttt{default} \multimap !^{(t+1).s.A}\texttt{box}(-,-) \otimes \bigotimes_{p\in suc(A)} !^{t.s.p}\texttt{P_RUN}\end{aligned}$$

Finally, if the parent of A is idle, A cannot perform any action and so its successors:

$$\begin{aligned}\texttt{ctr}(A,t) \stackrel{\texttt{def}}{=}\ & !^{t.s.A}\texttt{P_STOP} \multimap \texttt{stop-imm}(A,t)\& \\ & !^{t.s.A}\texttt{P_RUN} \multimap \texttt{decide}(A,t)\& \\ & !^{t.s.A}\texttt{P_IDLE} \multimap \forall n,m.(!^{t.s.A}\texttt{box}(n,m) \multimap !^{(t+1).s.A}\texttt{box}(n,m)) \\ & \otimes \bigotimes_{p\in suc(A)} !^{t.s.p}\texttt{P_IDLE}\end{aligned}$$

Encoding of States. As we have shown, the state of the system is represented in SELL by the predicate $\texttt{box}(\cdot)$. Then, a state S of a program P is encoded as:

$$[\![S]\!]_t = \bigotimes_{p\in S} !^{t.s.p}\texttt{box}(t_s(p), t_e(p)) \otimes \bigotimes_{p\in P\setminus S} !^{t.s.p}\texttt{box}(-,-)$$

Functions c_s and c_e are in Definition 1. Note that any $p \in P \setminus S$ corresponds to a TO that has not already started.

Encoding the Environment. The encoding requires that, at any time, it is possible to detect whether a given (external) event happened or not. Hence, the most general environment can be defined as $\texttt{env} \stackrel{\texttt{def}}{=} \Cap l : TI(\bigotimes_{m\in \texttt{M}} !^l(m \oplus m^\perp))$.

Recall that $t.i \preceq TI$ (see Figure 4). The universal quantification on subexponentials "$\Cap l : TI$" says that the formula $!^l(m \oplus m^\perp)$ is available in any time-unit (more precisely, in any subexponential of the form $t.i$). Here $m \oplus m^\perp$ means that either m was detected or not.

Encoding the program. A ReactiveIS program is encoded as the formula

$$[\![P]\!] = !^{0+}\Cap l : TP.!^l(\bigotimes_{p\in P} \texttt{ctr}(p,l))$$

Intuitively, the subexponential "0+" (along with the universal quantification "$\Cap l : TP$") allows us to copy, as many times as needed, the definition of the TOs in each time-unit.

Correctness of the Encoding

Let us recall some relevant concepts of SELL that will be important to understand the adequacy result presented below. The details can be found in [10]. SELL connectives are separated into two classes, the *negative* ones: $\multimap, \&, \top, \forall, \Bbbn$ and the *positive* ones: $\otimes, \oplus, \exists, \Bbbw, !, 1$. The polarity of non-atomic formulas is inherited from its outermost connective (e.g., $F \multimap G$ is a negative formula while $F \otimes G$ is a positive one) and positive bias is assigned to atomic formulas.

The focused proof system [2] of SELL [10] considers four kind of sequents:

(i) $[\mathcal{K} : \Gamma], \Delta \longrightarrow \mathcal{R}$ is an unfocused sequent. The meaning of the context $[\mathcal{K} : \Gamma]$ will be clear soon.
(ii) $[\mathcal{K} : \Gamma] -_F\!\!\rightarrow$ is a sequent focused on the right.
(iii) $[\mathcal{K} : \Gamma] \xrightarrow{F} G$ is a sequent focused on the left.
(iv) $[K : \Gamma], \Delta \longrightarrow [F]$ is a sequent representing the end of the negative phase.

As a matter of example, consider the following proof rules for multiplicative ($\otimes$) and additive ($\&$) conjunction, linear implication ($\multimap$), additive disjunction ($\oplus$) and the first-order quantifiers ($\forall, \exists$):

Negative Phase

$$\frac{[\mathcal{K} : \Gamma], \Delta, F, G \longrightarrow \mathcal{R}}{[\mathcal{K} : \Gamma], \Delta, F \otimes G \longrightarrow \mathcal{R}}\ \otimes_L \qquad \frac{[\mathcal{K} : \Gamma], \Delta, F \longrightarrow G}{[\mathcal{K} : \Gamma], \Delta \longrightarrow F \multimap G}\ \multimap_R \qquad \frac{[\mathcal{K} : \Gamma], \Delta \longrightarrow G[x_e/x]}{[\mathcal{K} : \Gamma], \Delta \longrightarrow \forall x.G}\ \forall_R$$

$$\frac{[\mathcal{K} : \Gamma], \Delta \longrightarrow F \quad [\mathcal{K} : \Gamma], \Delta \longrightarrow G}{[\mathcal{K} : \Gamma], \Delta \longrightarrow F \,\&\, G}\ \&_R \qquad \frac{[\mathcal{K} : \Gamma], \Delta, F \longrightarrow \mathcal{R} \quad [\mathcal{K} : \Gamma], \Delta, H \longrightarrow \mathcal{R}}{[\mathcal{K} : \Gamma], \Delta, F \oplus H \longrightarrow \mathcal{R}}\ \oplus_L$$

The proof rule $\exists_R$ is similar to $\forall_L$ and x_e is assume to be fresh. First notice that the negative connectives have invertible *right* rules, while the positive connectives have invertible *left* rules. For instance, consider the rule $\forall_R$: the choice of the name used for the eigenvariable x_e is not important for provability, as long as it is fresh. Hence, in a negative phase of the proof, no *backtracking* on the selection of inference rules is necessary. Moreover, without loosing provability, we can eagerly introduce all the negative non-atomic formulas on the right and all the positive non-atomic formulas on the left. Such part of the proof is represented by sequents of the shape (i) above.

A positive phase begins by choosing a formula on which to focus, enabling sequents of the forms (ii) or (iii). Let us introduce some of the proof rules that belong to this phase:

Positive Phase

$$\frac{[\mathcal{K} : \Gamma] -_{G_i}\!\!\rightarrow}{[\mathcal{K} : \Gamma] -_{G_1 \oplus G_2}\!\!\rightarrow}\ \oplus_{R_i} \qquad \frac{[\mathcal{K} : \Gamma] \xrightarrow{F_i} G}{[\mathcal{K} : \Gamma] \xrightarrow{F_1 \& F_2} G}\ \&_{L_i} \qquad \frac{[\mathcal{K} : \Gamma] \xrightarrow{F[t/x]} G}{[\mathcal{K} : \Gamma] \xrightarrow{\forall x.F} G}\ \forall_L$$

The rule $\oplus_R$ belongs to the positive phase since we have to chose between G_1 and G_2. Similarly, in Rule $\forall_L$, we need to decide on the term t. Note also that in the above rules, the focusing is not lost and the proof must continue decomposing

the selected formula (for instance, in $\oplus_R$, the focusing –on the right– persists on G_i). This procedure continues until one is focused either on a negative formula on the right or a positive formula on the left. This point marks the end of the positive phase and another negative phase starts.

We can classify the formulas produced by our encoding as guards (G) and programs (P):

$$\begin{array}{l} G ::= !^s.A \mid G \otimes G \mid G \oplus G \mid \exists x.G \\ P ::= !^s.A \mid P \otimes P \mid P \,\&\, P \mid G \multimap P \mid \forall x.P \end{array} \tag{1}$$

Guards will appear on the right hand side of the sequent while programs will appear on the left hand side. This separation of formulas is important to prove the adequacy result. The idea is that once we are focused on a formula representing a TO (i.e., a P-formula), we have to completely decompose it in a positive phase of the proof. In the end of this phase, what we observe is that the state changed exactly as the operational rules dictate.

Theorem 3 (Adequacy). *Let P be a* ReactiveIS *program. Then, $\langle S, t\rangle \overset{I,O}{\Longrightarrow}_P \langle S', t+1\rangle$ iff the sequent $[\![P]\!], [\![S]\!]_t, [\![I]\!]_t \longrightarrow [\![S']\!]_{t+1} \otimes [\![O]\!]_t$ is provable in SELL.*

Proof. We show that the introduction of any formula, following the focused discipline, corresponds exactly to applying one of the operational rules. Consider that we focus on a $\texttt{ctr}(A,t)$ formula obtaining the derivation below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi}{\Gamma'' \xrightarrow{G}} \quad \dfrac{\Psi}{\Gamma''' \multimap_P \rightarrow}}{\Gamma' \xrightarrow{G \multimap P}} \multimap_L}{\Gamma \xrightarrow{\texttt{ctr}(A,t)} H} \&_L$$

The main connective in `ctr` is "&" and the focusing persists in one of the choices. In this case, $G = !^{t.s.A}G'$, where G' can be one of the predicates `P_STOP`, `P_RUN` or `P_IDLE`. Since the subexponential $t.s.A$ is unrelated to the others, the derivation Π must finish by proving G' from the context Γ''' that only contains facts about `A` (due to the promotion rule $!^s{}_R$).

Derivation Ψ on the right hand side proceeds similarly. Here P can be a P-formula (see Equation (1)) of the shape $P \,\&\, P$, $G \multimap P$ or $\forall x.P$. In all these cases, we have negative connectives (on the left) that have to be introduced in a positive phase. Hence, the focusing persists on P and we do not have other choice that continuing decomposing this formula.

Consider the case where P is the formula $\texttt{str}(A,t)$. We then observe

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1}{\Gamma' \multimap_{G_1} \rightarrow} \quad \dfrac{\Psi_1}{\Gamma'' \xrightarrow{P_1}}}{\Gamma \xrightarrow{G_1 \multimap P_1}} \&_L}{\Gamma \xrightarrow{\texttt{str}(A,t)}}$$

Here G_1 can be the formula `condition` or `default`. In any case, this formula if a G-Formula (see Equation (1)). Therefore, the focusing persists and we end up in a situation similar to Π above.

The formula P_1 on the right hand side takes the form $!^s A_1 \otimes \ldots \otimes !^s A_n$ where A_i is a predicate. Since "$\otimes$" and $!^s$ on the left has to be introduced in the negative phase, what we observe is that we lost focusing and, in a negative phase, all the formulas of the shape `msg`$(\cdot)$ and `box`$(\cdot)$ are added into the context. Thus, in a flip of the polarity, we observe that the state is modified exactly as the operational rules dictate. $\square$