The Smix synchronous multimedia language: Operational semantics and coroutine implementation

Guilherme F. Lima[1], Christiano Braga[2], and Edward Hermann Haeusler[1]

[1] PUC-Rio, Rio de Janeiro, Brazil
{glima,hermann}@inf.puc-rio.br
[2] UFF, Niterói, Brazil
cbraga@ic.uff.br

Abstract. Smix is a domain-specific language for the construction of interactive multimedia presentations. Its programs describe how media objects (texts, images, videos, etc.) should be presented and how external events, such as the passage of time or user interaction, affect their presentation. What distinguishes Smix from similar high-level multimedia languages, such as NCL, SMIL, and HTML, is first is simplicity: the language has only three main concepts (media object, action, and link) which can nonetheless be used to program complex multimedia applications. The second distinguishing characteristic of Smix is its synchronous, deterministic semantics, which induces a precise notion of logical time. In this paper, we introduce the Smix language, present two versions of its synchronous semantics, equational and linear, both in big-step operational style, and discuss a novel, straightforward implementation of its linear semantics using Lua coroutines.

1 Introduction

Smix [10] (Synchronous Mixer) is a domain-specific language for the construction of interactive multimedia presentations. Its programs use synchrony relations (links) to describe how media objects (texts, images, videos, etc.) should be presented and how external events, such as the passage of time or user interaction, affect their presentation.

There are two characteristics that distinguishes Smix from similar high level multimedia languages, such as NCL, SMIL, and HTML.[3] The first one is simplicity. Smix has only three main concepts, namely, media, action, and link, which can nonetheless be used to program complex multimedia applications. A simpler language implies in a simpler semantics and, consequently, a simpler implementation. NCL, SMIL, and HTML, in contrast, are huge languages with numerous concepts and constructions to represent them. Despite its simplicity, most NCL concepts can be easily simulated in Smix, as discussed in [10]; in fact, the Smix language was deliberately designed to serve as an abstraction layer (the language of a multimedia virtual machine) for the implementation of other higher-level multimedia languages, in particular, Plain Smix (a syntactically richer dialect of Smix), NCL, and to a lesser extend, SMIL.

[3] NCL is the standard declarative language for interactive applications in the Brazilian digital terrestrial television system [1] and an ITU-T recommendation for IPTV applications [8]. SMIL is a widely adopted W3C recommendation [16] for interactive multimedia presentations. And HTML is a W3C recommendation [17] (and core Web technology) for typesetting hyperlinked text together with images, and more recently, audio and video.

The second distinguishing characteristic of Smix is its deterministic, synchronous semantics, which gives its programs a precise notion of logical time. The semantics of NCL, SMIL and HTML, in contrast, is notoriously complex and obscure, especially in relation to time [10]. Even on a logical level, these languages treat time as something external to the system. Its representation and manipulation can be influenced by physical phenomena, such as processing or communication delays, which are unpredictable or implementation dependent, and which can thus lead to nondeterminism and dyssynchrony. Smix, on the other hand, is a synchronous language with a semantics that guarantees determinism and logical correctness. By calling it synchronous, we mean that its programs operate under the *synchronous hypothesis* [5], i.e., that they can be viewed as input-driven systems whose reactions are instantaneous. The synchronous hypothesis induces a precise notion of logical time in which the only relevant concepts are those of simultaneity and precedence between events.

In [15] the authors propose a rewriting-logic semantics for NCL, and in [14] the author proposes an authoring language-independent model for multimedia documents. There are other proposals of formal semantics for NCL [12] and similar proposals for SMIL [4]. Most of these works, however, are concerned not with the implementation of interpreters (which is the main goal of Smix) but with static validation of program properties, usually within a larger system of user-guided verification. Their models tend to be complex and impractical, especially if real-time performance is needed.

In this paper, we focus on primarily the formalization of the synchronous semantics of Smix, and present two versions of it: the equational semantics and the linear semantics. Both versions follow the operational approach to semantics [13]; the particular style used is that of big-step (or natural) operational semantics [9]. Both formalizations were inspired by the formal semantics of the synchronous language Esterel [3,2], and as such are only concerned with the description of a single program reaction.

The rest of the paper is organized as follows. In Section 2, we introduce the Smix language and discuss the intuitive behavior of its programs. In Section 3, we present the equational semantics, which formalizes this intuitive behavior. The problem with the equational semantics is that it does not guarantee termination in bounded time, which violates the synchronous hypothesis. In Section 4, we present the linear semantics that solves this problem by adopting a linear format for programs, which replaces the equational format, and in which reactions always execute in bounded time. In practice, before executing a program, the Smix interpreter "linearizes" it, i.e., converts it from the equational to the linear format. In Section 5, we present a simple implementation of the linear semantics in a Lua [7] multimedia library augmented with coroutines (Section 5). Finally, in Section 6 we draw our conclusions and point out future work.

2 The Smix language

Smix is a high-level declarative language for the construction of multimedia presentations. Its goal is to offer simple but expressive abstractions for the precise representation of complex audiovisual ideas. A Smix program is a set of media object declarations together with a sequence of links. A media object is a presentation atom (e.g., image, audio, video, etc.) and has associated with it an identifier, a content, a state, a time, and a property table.

The identifier uniquely identifies object in the program. The content is a possibly empty sequence of audiovisual samples. The state is either "occurring" (playing), "paused", or "stopped". The time is the number of clock ticks to which the object was exposed while in state occurring. And the property table maintains the object properties—their value determine the characteristics of the object's presentation, e.g., the value of property "transparency" determines the transparency applied to its visual samples.

In Smix, media objects are manipulated by actions. There are five possible actions: start (▷), stop (□), pause (▯▯), seek (⏩), and attribution (○). The first three actions, start, stop, and pause, manipulate the object's state; the last two, seek and set, manipulate the object's time and property table. Actions have the general form (predicate ? target : argument), where the predicate is a propositional logic formula involving the state, time, or property values of media objects, the target specifies the operation (▷, □, ▯▯, ⏩, or ○) and main operand (media object or property) of the action, and the argument is an extra operand (expression) required by seek and set actions.

The execution of an action is conditioned by the validity of its predicate. To evaluate an action, the interpreter (more precisely, the language kernel) first evaluates its predicate. If it is false, the action is discarded; otherwise, if it is true, the kernel proceeds to execute the action: it evaluates the extra argument (if any) and tries to execute the specified operation with the given operands. When writing actions, we often omit the predicate, question mark, and parentheses when the predicate is tautological (always true). Thus (i) an action of the form $\triangleright x$, read "start x", when executed, puts x in state occurring; (ii) an action of the form ▯▯x, read "pause x", puts x in state paused; (iii) an action of the form $\square x$, read "stop x", puts x in state stopped; (iv) an action of the form ⏩$x : e$, read "seek x by e", advances the playback time of x by the number to which expression e evaluates; and (v) an action of the form $\bigcirc x.u : e$, read "set $x.u$ to e", stores into property u of x the value to which expression e evaluates.

A Smix program consists of two parts: a set of media object declarations and a sequence of links. A media object declaration associates an object identifier with a property initialization table. A link is a synchrony relation of the form $a \rightarrow a_1 a_2 \ldots a_n$, which establishes that whenever some action with target a is executed, actions $a_1, a_2, \ldots, a_n$ shall also be executed, in this order. The action target a on the left-hand side of symbol $\rightarrow$ is called the head of the link, and the action sequence $a_1 a_2 \ldots a_n$ on its right-hand side is called the tail of the link.

Example. To make matters concrete, consider the following Smix program:

$$\triangleright\lambda \rightarrow \triangleright x$$
$$\triangleright x \rightarrow \triangleright y \, \square z$$
$$\triangleright y \rightarrow \triangleright z$$
$$\square x \rightarrow \square\lambda$$

This program has four links which operate on four media objects: the ordinary objects x, y, and z, and the implicit object lambda (λ) which stands for the program itself. The first link establishes that when the program starts, media object x shall be started; the second link establishes that whenever x starts, object y shall be started and object z shall be stopped; the third link establishes that whenever y starts, object z shall be started; and the fourth link establishes that when x stops the whole program shall be stopped.

The equational and linear semantics discussed in Sections 3 and 4 and are only concerned with the description of a single program reaction (input-output cycle). Given some input action a received from the environment, they determine how the execution of action a affects the kernel memory (state, time and properties of media objects) and the actions $a_1, a_2, \ldots, a_n$ that are to be triggered internally in response to a, and emitted back to the environment at the end of the reaction.

3 The equational semantics

Smix has the following syntactic sets: integers $n \in \mathrm{N}$; truth values $t \in \mathrm{T} = \{\top, \bot\}$; media object identifiers $x, y, z \in \mathrm{Media}$; property identifiers $u, v \in \mathrm{Prop}$; expressions $e \in \mathrm{Expr}$; predicates $p \in \mathrm{Pred}$; action atoms $a \in \mathrm{ActAtom}$; action sequences $\alpha \in \mathrm{ActSeq}$; and link sequences (or programs) $L, P \in \mathrm{LinkSeq}$. Its abstract syntax is defined as follows:

$$
\begin{aligned}
e \in \mathrm{Expr} &::= n \mid \mathrm{state}(x) \mid \mathrm{time}(x) \mid \mathrm{prop}(x,u) \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 \div e_2 \\
p \in \mathrm{Pred} &::= \top \mid \bot \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid \neg p_1 \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \\
a \in \mathrm{ActAtom} &::= (p\,?\,\triangleright x) \mid (p\,?\,\blacksquare x) \mid (p\,?\,\square x) \mid (p\,?\,\twoheadrightarrow x{:}e) \mid (p\,?\,\circ x.u{:}e) \\
\alpha \in \mathrm{ActSeq} &::= \varepsilon \mid a\alpha_1 \\
L \in \mathrm{LinkSeq} &::= \varepsilon \mid \triangleright x \to \alpha L_1 \mid \blacksquare x \to \alpha L_1 \mid \square x \to \alpha L_1 \mid \twoheadrightarrow x \to \alpha L_1 \mid \circ x.u \to \alpha L_1
\end{aligned}
$$

The program state is represented by a media memory, i.e., a total function θ that maps a media object identifier x to a memory cell $\langle s, n, \rho \rangle$, where $s \in \{\triangleright, \blacksquare, \square\}$ is the object state, $n \in \mathrm{N}$ is its time, and $\rho\colon \mathrm{Prop} \to \mathrm{N}$ is a total function that represents its property table. We write $\mathcal{M}$ for the set of all media memories, ϕ for the empty memory cell $\langle \square, 0, \rho_0 \rangle$, where ρ_0 is the table in which all properties have value 0, and Φ for the empty memory, i.e., the one in which all cells are empty.

Memory cells can be read and written. Given a memory θ and a media object x, we write $\theta(x)$ for the cell of x in θ and $\theta[x := X]$ for the memory obtained by replacing $\theta(x)$ by X. We write $\theta_s(x)$, $\theta_t(x)$, $\theta_\rho(x,u)$ for the state, time, and value of property u of x in θ, and $\theta_s[x := s]$ for the memory obtained by replacing $\theta_s(x)$ by s, $\theta_t[x \mathrel{+}= n]$ for the memory obtained by incrementing $\theta_t(x)$ by n, and $\theta_\rho[x.u := n]$ for the memory obtained by replacing $\theta_\rho(x,u)$ by n.

Finally, to access the links of a program, we define the link function ℓ that receives as arguments the program P and an action atom a, and returns the action sequence α associated with the execution of a in P ($\tau(a)$ denotes the target of action a):

$$
\begin{aligned}
\ell(\varepsilon, a) &= \varepsilon \\
\ell(a' \to \alpha L, a) &= \begin{cases} \alpha \ell(L, a) & \text{if } \tau(a) = a' \\ \ell(L, a) & \text{otherwise}\,. \end{cases}
\end{aligned}
$$

Evaluation of equational programs. The evaluation of action sequences is determined by the relation $\Rightarrow$ such that $\langle \alpha, P, \theta \rangle \Rightarrow \theta'$ iff action sequence α when executed over program P in memory θ evaluates to the updated memory θ'. Since program P remains fixed throughout the evaluation, we use the notation $\langle \alpha, \theta \rangle \Rightarrow \theta'$, with references to an

implicit program P made when necessary. The relation $\Rightarrow$ is defined inductively in terms of the link function and the relations for evaluation of expressions and predicates (whose definition we deliberately omit) by the following eleven rules.

$$\langle \varepsilon, \theta \rangle \Rightarrow \theta \tag{R_ε}$$

$$\frac{\langle \mathrm{state}(x) \neq \triangleright \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle \ell(P, \triangleright x)\alpha, \theta_s[x := \triangleright] \rangle \Rightarrow \theta'}{\langle (p\,?\triangleright x)\alpha, \theta \rangle \Rightarrow \theta'} \tag{$R^+_\triangleright$}$$

$$\frac{\langle \mathrm{state}(x) \neq \triangleright \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\triangleright x)\alpha, \theta \rangle \Rightarrow \theta'} \tag{$R^-_\triangleright$}$$

$$\frac{\langle \mathrm{state}(x) = \triangleright \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle \ell(P, \blacksquare x)\alpha, \theta_s[x := \blacksquare] \rangle \Rightarrow \theta'}{\langle (p\,?\blacksquare x)\alpha, \theta \rangle \Rightarrow \theta'} \tag{$R^+_\blacksquare$}$$

$$\frac{\langle \mathrm{state}(x) = \triangleright \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\blacksquare x)\alpha, \theta \rangle \Rightarrow \theta'} \tag{$R^-_\blacksquare$}$$

$$\frac{\langle \mathrm{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle \ell(P, \square x)\alpha, \theta[x := \phi] \rangle \Rightarrow \theta'}{\langle (p\,?\square x)\alpha, \theta \rangle \Rightarrow \theta'} \tag{$R^+_\square$}$$

$$\frac{\langle \mathrm{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\square x)\alpha, \theta \rangle \Rightarrow \theta'} \tag{$R^-_\square$}$$

$$\frac{\langle \mathrm{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle e, \theta \rangle \Rightarrow n \quad \langle \ell(P, \twoheadrightarrow x)\alpha, \theta_t[x \mathrel{+}= n] \rangle \Rightarrow \theta'}{\langle (p\,?\twoheadrightarrow x{:}e)\alpha, \theta \rangle \Rightarrow \theta'} \tag{$R^+_\twoheadrightarrow$}$$

$$\frac{\langle \mathrm{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\twoheadrightarrow x{:}e)\alpha, \theta \rangle \Rightarrow \theta'} \tag{$R^-_\twoheadrightarrow$}$$

$$\frac{\langle \mathrm{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle e, \theta \rangle \Rightarrow n \quad \langle \ell(P, \circ x.u)\alpha, \theta_\rho[x.u := n] \rangle \Rightarrow \theta'}{\langle (p\,?\circ x.u{:}e)\alpha, \theta \rangle \Rightarrow \theta'} \tag{$R^+_\circ$}$$

$$\frac{\langle \mathrm{state}(x) \neq \square \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha, \theta \rangle \Rightarrow \theta'}{\langle (p\,?\circ x.u{:}e)\alpha, \theta \rangle \Rightarrow \theta'} \tag{$R^-_\circ$}$$

By rule R_ε the empty sequence ε does nothing and leaves the memory unchanged.

By rule $R^+_\triangleright$, if the first action of the sequence is $(p\,?\triangleright x)$ and if it can be executed in state θ, i.e., if object x is in state paused or stopped and predicate p evaluates to true in θ, the configuration evaluates to the result of evaluating sequence $\ell(P, \triangleright x)\alpha$ in $\theta_s[x := \triangleright]$; otherwise, by rule $R^-_\triangleright$, the configuration evaluates to the result of evaluating α in θ.

Rules $R^+_\blacksquare$, $R^-_\blacksquare$, and $R^-_\square$ operate similarly. If the first action of the sequence can be executed, x transitions to the corresponding state and the links that depend on the action target are triggered; otherwise, the action is dropped and the next action of the sequence is considered. Rule $R^+_\square$ is also similar, but besides transitioning x to state stopped, it replaces the cell of x in θ by the empty cell ϕ, which resets x's state, time, and properties.

By rule $R^+_\twoheadrightarrow$, if the first action of the sequence is $(p\,?\twoheadrightarrow x{:}e)$ and if it can be executed in θ, x's playback time is incremented by the number to which expression e evaluates in θ, and the links of program P that depend on target $\twoheadrightarrow x$ are triggered; otherwise, by rule $R^-_\twoheadrightarrow$, action $(p\,?\twoheadrightarrow x{:}e)$ is dropped and the next action of the sequence is considered. By definition of memory writes, the playback time of x is reset to 0 if $\theta_t(x) + n < 0$; thus the resulting playback time is always a nonnegative integer.

By rule $R^+_\circ$, if the first action of the sequence is $(p\,?\circ x.u{:}e)$ and if it can be executed, property u of x is set to the number to which expression e evaluates in θ, and the links of program P that depend on target $\circ x.u$ are triggered; otherwise, by rule $R^-_\circ$, action $(p\,?\circ x.u{:}e)$ is dropped and the next action of the sequence is considered.

Determinism and non-termination. Theorem 1 establishes that the evaluation of action sequences is deterministic. The proof follows by induction on the structure of derivations.

Theorem 1 (Determinism). *For all* $\alpha \in \text{ActSeq}$, θ, θ_1, $\theta_2 \in \mathcal{M}$, *if* $\langle \alpha, \theta \rangle \Rightarrow \theta_1$ *and* $\langle \alpha, \theta \rangle \Rightarrow \theta_2$ *then* $\theta_1 = \theta_2$.

Theorem 2 establishes that the evaluation of action $\triangleright x$ in the empty memory Φ with $P = \triangleright x \rightarrow \Box x \triangleright x$ does not converge. The proof follows by contradiction on the assumption of minimality of a hypothetical derivation of $\langle (\top \,?\triangleright x), P, \Phi \rangle \Rightarrow \theta$.

Theorem 2. *Let* $P = \triangleright x \rightarrow (\top \,?\Box x)(\top \,?\triangleright x)$. *Then there is no* $\theta \in \mathcal{M}$ *such that* $\langle (\top \,?\triangleright x), P, \Phi \rangle \Rightarrow \theta$.

The above theorem implies that, under the equational semantics, the computation of reactions may not terminate in a finite number of steps, which violates the synchronous hypothesis. Similar problems occur in related languages, e.g., the problem of cyclic dependencies in SMIL's timegraph [16] (the structure used by the SMIL interpreter to control the presentation), or that of causality cycles in Esterel [2]. Here the problem is caused by infinite feedback loops in link evaluation: a link (or group of links) triggers its reevaluation endlessly. A common approach to tackle such tight loops is to impose a restriction that breaks them. For example, we could establish an upper bound to the number of times the same link or action can execute during a reaction. Though such restrictions are reasonable, we follow a more flexible path. Instead of adopting a particular a priori restriction, we introduce a linear format for programs in which links and action sequences are replaced by equivalent linear programs that always terminate.

4 The linear semantics

The abstract syntax of linear programs is mostly identical to that of equational programs presented in Section 3. The only difference is the substitution of sets ActSeq and LinkSeq by the set ActLine of linear programs defined as follows:

$$\alpha \in \text{ActLine} ::= \varepsilon \mid a[\alpha_1]\alpha_2$$

Here metavariable α is assumed to range over ActLine. Though the same metavariable is used to denote action sequences (members of ActSeq), care is taken not to mix the uses so that the correct denotation can always be inferred from the context.

The linearization procedure σ we adopt takes as input an equational program P and an action a and outputs a linear program α that represents the evaluation of a in P. The procedure σ is defined in terms of the graph of program P, which is built by interpreting its links as an adjacency list. For instance, Figure 1 depicts a Smix program and its corresponding graph. A loop in the graph indicates the possibility of a tight loop during reaction evaluation, but it does not guarantee that it will occur—its occurrence depends on the contents of the evaluation stack and media memory, both of which cannot be known statically.

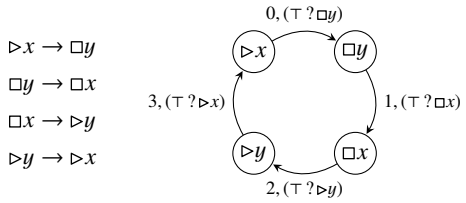

Fig. 1. A Smix program and its corresponding graph.

Given some program P and action a, procedure σ starts at the node representing the target of action a and proceeds in depth-first fashion traversing (marking) each reachable arc at most once. Its result is the linear program that implements the execution a in P. The procedure's running time is bounded to the number of arcs reachable from its point of departure; its time complexity is thus $O(n)$ where n is size of program P.

By applying σ to the program of Figure 1 with an input action $(\top\ ?\triangleright x)$, we get the linear program $\triangleright x[\square y[\square x[\triangleright y]]]$. This program encodes the dependencies between actions on the original equational program. To evaluate it, the kernel reads its leftmost action, $\triangleright x$, and tries to execute it. If it succeeds, in this case, if x can transition to state occurring in θ, it proceeds to evaluate the subprogram that depends on $\triangleright x$, namely, the subprogram immediately following it in square brackets, $\square y[\square x[\triangleright y]]$. Otherwise, it skips the brackets altogether and proceeds to evaluate the next subprogram, ε in this case. The kernel continues until there are no actions left to be executed.

Evaluation of linear programs. The evaluation of linear programs is given by the relation $\Rightarrow$ such that $\langle \alpha, \theta \rangle \Rightarrow \theta'$ iff linear program α when executed in memory θ evaluates to an updated memory θ'. Relation $\Rightarrow$ is defined inductively in terms of the relations for evaluation of expressions and predicates by the following rules. (Here we show only the rules for the evaluation of programs whose first action is a start action; the rules for the remaining actions and for the empty program are similar—they are analogous to their counterparts in the equational semantics.)

$$\frac{\langle \text{state}(x) \neq \triangleright \wedge\ p, \theta \rangle \Rightarrow \top \quad \langle \alpha_1 \alpha_2, \theta_s[x := \triangleright] \rangle \Rightarrow \theta'}{\langle (p\ ?\triangleright x)[\alpha_1]\alpha_2, \theta \rangle \Rightarrow \theta'} \qquad (R^+_{\triangleright})$$

$$\frac{\langle \text{state}(x) \neq \triangleright \wedge\ p, \theta \rangle \Rightarrow \bot \quad \langle \alpha_2, \theta \rangle \Rightarrow \theta'}{\langle (p\ ?\triangleright x)[\alpha_1]\alpha_2, \theta \rangle \Rightarrow \theta'} \qquad (R^-_{\triangleright})$$

Determinism and termination. Theorem 3 establishes that the evaluation of linear programs is deterministic. The proof follows by induction on the structure of derivations.

Theorem 3 (Determinism). *For all* $\alpha \in \text{ActLine}$, θ, θ_1, $\theta_2 \in \mathcal{M}$, *if* $\langle \alpha, \theta \rangle \Rightarrow \theta_1$ *and* $\langle \alpha, \theta \rangle \Rightarrow \theta_2$ *then* $\theta_1 = \theta_2$.

Theorem 4 establishes that the evaluation of linear programs always terminates. Its proof follows by induction on the structure of programs and depends on a lemma that establishes that $\langle \alpha_1 \alpha_2, \theta \rangle \Rightarrow \theta''$ iff $\langle \alpha_1, \theta \rangle \Rightarrow \theta'$ and $\langle \alpha_2, \theta' \rangle \Rightarrow \theta''$, for some θ'.

Theorem 4 (Termination). *For all* $\alpha \in \text{ActLine}$ *and* $\theta \in \mathcal{M}$, *there is a* $\theta' \in \mathcal{M}$ *such that* $\langle \alpha, \theta \rangle \Rightarrow \theta'$.

A consequence of Theorem 4 is the Turing-incompleteness of the computational model of linear Smix programs. One requirement for Turing-completeness is the ability to express indefinite iteration, but Theorem 4 restricts this ability, so the resulting model is not Turing-complete. This means that there are computable functions which cannot be expressed by linear Smix programs. That said, Smix's model is intentionally restricted: it aims to ease the description of interactive multimedia presentations, as opposed to the description of general algorithms. Moreover, if general computing functions are required, one can resort to external scripts, which can be embedded in the program as media objects containing Lua code.

Finally, note that the evaluation relation for linear programs determines a natural equivalence relation on ActLine: programs α_1 and α_2 are equivalent, in symbols $\alpha_1 \sim \alpha_2$, iff they evaluate to the same final memory θ' when fed with the same initial memory θ. This definition of equivalence gives rise to program reduction techniques which can be used to optimize programs. Equivalence results and the detailed proofs of the previous theorems can be found in [10].

5 Coroutine interpretation

The original implementation of Smix [10] has two parts: the language kernel, which is simply a realization of the linear semantics, and the multimedia engine, which take kernel's commands and renders the corresponding multimedia presentation. These parts are kept in isolated modules that communicate asynchronously by exchanging messages (actions). Though this design works reasonably well, an even simpler implementation is possible: we can convert (or interpret) the Smix program into a Lua script that uses the multimedia engine's synchronous API plus Lua coroutines to realize the program logic. We now describe this alternative implementation in detail.

Smix's multimedia engine code consists of a single C library, called LibPlay[4], which is built on top of GStreamer [6], a free/open-source framework for multimedia. The Lua binding of LibPlay is called LuaPlay, and has two main concepts: scene and media. A scene represents an OS-level window with audio and video output. And a media represents a media object, which is analogous to a Smix media object. The scene API consists of the following functions: (i) *new*, which creates it, (ii) *get* and *set* which manipulate its properties, (iii) *receive* which blocks awaiting for a given event, and (iv) *quit* which quits the scene. Similarly, the media API consists of the functions (i) *new* which creates a media in a given scene, (ii) *get* and *set* which manipulates the media properties, and (iii) *start*, *pause*, *stop* and *seek* which manipulates the media state and playback time.

The scene and media APIs we are considering here are synchronous: all its calls are immediately effectuated and, with exception of scene's *receive* call, execute in no (logical) time. The only call that actually "consumes" time is *receive*; in fact, it is only during this call that the engine produces audiovisual samples, and it does this until an event that matches the mask passed to *receive* is generated. Currently, LuaPlay API supports three types of events: clock ticks, user interactions (keyboard and mouse) and media object state changes.

[4] `https://github.com/TeleMidia/LibPlay`

Using the previous API, we can easily construct simple applications that wait for a single event before doing something. But as soon as we need to wait for more than one event things get complicated. There are basically two approaches to deal with the problem of awaiting on multiple events (conditions). The traditional solution is to use callbacks—we could call *receive* in a loop, passing each received event to the registered callbacks. The problem with this solution is that the program logic is "lost" in the callbacks. The alternative solution, and the one we adopt here, is to implement a parallel operation that creates new program trails dynamically. Under this approach, to wait for two events we simply create two trails and block them on the corresponding events.

In LuaPlay, this parallel operator is the scene function *par*: it creates a trail for each function received as argument and terminates the parallel composition as soon as one of them ends. In practice, we use Lua coroutines to implement the parallel composition. The *par* call creates a coroutine to represent the parallel composition of the given functions. It wraps each function into a coroutine itself and then execute these child coroutines, one at a time. If all of them yield awaiting on some condition, the composition itself yields awaiting on the combined condition. Otherwise, if one of them terminates, the composition terminates, which causes the termination of its child trails. (If *par* calls are nested, only the topmost call calls the real "await", i.e., the scene's *receive* function.)

Using LuaPlay's parallel operator *par* and an *await* operator, which is simply the coroutine yield call, we can easily implement Smix programs: the program itself is a single *par* call and each of its links is a trail that waits in a loop for the link condition (its head) and executes the corresponding linear program whenever it is awaken. Figure 2 presents the LuaPlay program that implements the example Smix program discussed at the end of Section 2. Finally, note that using this technique we can either compile Smix programs into LuaPlay programs or interpret them directly, i.e., we can write an *eval* function which takes a Smix program, builds and returns a function that is the corresponding LuaPlay program (the main trail of the *par* call).

```
1  scene:par {
2    function ()
3      while true do
4        await {type='start', media=λ}
5        exec (σ(P,⊳λ))
6      end end,
7    function ()
8      while true do
9        await {type='start', media=x}
10       exec (σ(P,⊳x))
11     end end,
12   function ()
13     while true do
14       await {type='start', media=y}
15       exec (σ(P,⊳y))
16     end end,
17   function ()
18     while true do
19       await {type='stop', media=x}
20       exec (σ(P,□x))
21     end end
22 }
```

Fig. 2. Coroutine version of the example Smix program discussed at the end of Section 2.

6 Conclusion

In this paper, we presented the Smix language, discussed two versions its synchronous semantics, equational and linear, and proposed a novel, straightforward implementation of its linear semantics using Lua coroutines. Though we discussed most Smix features, some of them (pinned actions, limited iteration, and asynchronous actions) were deliberately omitted. These omissions, however, do not affect the formalisms and results discussed in Sections 3 and 4, nor the coroutine implementation discussed in Section 5.

We are currently investigating a continuation semantics for the coroutine interpretation of Smix programs discussed in Section 5. Our goal, in this case, is not only to relate both semantics (Smix and continuations) but also to use the continuation semantics as a basis for developing an imperative (Esterel-like) synchronous multimedia language (whose engine is LuaPlay). Besides the continuation semantics, we are also investigating approaches for verifying the behavior of Smix programs "in time", i.e., for reasoning about a sequence of chained program reactions, each describing an instant. We intend to use for this purpose Petri-PDL [11], an extension of dynamic propositional logic for Petri nets, which would allow us to model the behavior of both the language kernel and the multimedia engine, and to do so stochastically.

References

1. ABNT NBR 15606-2: Digital Terrestrial TV — Data Coding and Transmission Specification for Digital Broadcasting — Part 2: Ginga-NCL for Fixed and Mobile Receivers: XML Application Language for Application Coding. ABNT, São Paulo, SP, Brazil (2007)
2. Berry, G.: The constructive semantics of pure Esterel: Draft version 3. Tech. rep., INRIA, Sophia-Antipolis, France (2002)
3. Berry, G., Gonthier, G.: The ESTEREL synchronous programming language: Design, semantics, implementation. Science of Computer Programming 19(2) (1992)
4. Gaggi, O., Bossi, A.: Analysis and verification of SMIL documents. Multimedia Systems 17(6) (2011)
5. Gamatié, A.: Designing Embedded Systems with the SIGNAL Programming Language. Springer New York, New York, NY, USA (2010)
6. GStreamer Developers: GStreamer: Open source multimedia framework. `http://gstreamer.freedesktop.org`, accessed November 9, 2016
7. Ierusalimschy, R.: Programming in Lua. Lua.org, 3rd edn. (2013)
8. ITU-T Recommendation H.761: Nested Context Language (NCL) and Ginga-NCL. ITU Telecommunication Standardization Sector, Geneva, Switzerland (November 2014)
9. Kahn, G.: Natural semantics. In: STACS 87: 4th Annual Symposium on Theoretical Aspects of Computer Science, 1987 Proceedings, LNCS, vol. 247 (1987)
10. Lima, G.F.: A synchronous virtual machine for multimedia presentations. Ph.D. thesis, Department of Informatics, PUC-Rio, Rio de Janeiro, RJ, Brazil (2015)
11. Lopes, B.: Extending Propositional Dynamic Logic for Petri Nets. Ph.D. thesis, Department of Informatics, PUC-Rio, Rio de Janeiro, RJ, Brazil (2014)
12. Picinin, D., Farines, J.M., Koliver, C.: An approach to verify live NCL applications. In: Proceedings of the 18th WebMedia, São Paulo, SP, Brazil, 15–18 October, 2012. ACM (2012)
13. Plotkin, G.D.: A structural approach to operational semantics. Tech. Rep. 19, Computer Science Departement, Aarhus University, Aarhus, Denmark (1981)
14. dos Santos, J.: Multimedia Document Validation Along its Life Cycle. Ph.D. thesis, Computing Institute, UFF, Niterói, RJ, Brazil (2016)
15. dos Santos, J., Braga, C., Muchaluat-Saade, D.C.: A rewriting logic semantics for NCL. Science of Computer Programming 107–108 (2015)
16. W3C: Synchronized multimedia integration language (SMIL 3.0). Recommendation, World Wide Web Consortium (December 2008)
17. W3C: HTML5: A vocabulary and associated APIs for HTML and XHTML. Recommendation, World Wide Web Consortium (October 2014)